MASTER IN COMPUTER SCIENCE

An experiment in agent development using non-monotonic reasoning

Ligny, Thomas; Marlier, Benoît

Award date:
2000

Faculté Universitaires Notre-Dame de la Paix
Institut d'informatique - Rue Grangagnage, 21
5000 NAMUR

An experiment in agent development using non-monotonic reasoning

Thomas Ligny and Benoît Marlier

Dissertation presented with the objective to obtain the degree of Master in Computer Science

Academic year 1999-2000

Promoter : Pierre-Yves Schobbens

Thanks

With much gratitude to Dr Mary-Anne Williams who assisted us in our work and who provided us the means to accomplish our research in the best way possible. She made our training period in the University of Newcastle, Australia very enriching and pleasant.

We would also like to thank Dr Pierre-Yves Schobbens for he has given us the opportunity to make a dissertation on a subject that is very interesting.

Benoit and Thomas

Résumé

Dans ce mémoire, nous faisons le point sur la discipline en plein développement des agents intelligents. Pour ce faire, nous avons opté pour une approche tant théorique que pratique. Nous exposons d'abord les principaux concepts relatifs aux agents intelligents et nous présentons ensuite une application basée sur ces concepts.

Dans un même temps, nous avons voulu revenir aux origines du concept d'agent intelligent en nous penchant plus spécifiquement sur le rôle de l'intelligence artificielle (IA) dans le développement des agents intelligents. Pour cela, nous présentons les bases théoriques d'une technique de l'IA (la logique non-monotone) et nous décrivons la manière dont nous l'avons intégré dans le raisonnement d'agents de notre application.

L'approche adoptée consiste à partir des besoins d'une application réelle et analyser si le paradigme des agents est apte à répondre à ces besoins.

Abstract

In this dissertation, we draw up an overview of a discipline in full development, namely the discipline of Intelligent Agents. We opted for a theoretical as well as a practical approach in order to accomplish our objective. First of all, we present the principal concepts relating to Intelligent Agents, followed by an application based on those concepts.

At the same time, we wanted to go back to the source of the concept 'Intelligent Agent'. More specifically, we did this by looking at the role of Artificial Intelligence (IA) in developing Intelligent Agents. To achieve this, we discuss the theoretical bases of a technique used in IA (non-monotonic logic) and we describe how we have integrated this technique in the reasoning of our agents.

In short, our approach looks at the needs of a real application and analyses whether the agent paradigm is capable of addressing those needs.

TABLE OF CONTENTS

GENERAL INTRODUCTION

Big changes are taking place in the area of information supply and demand. The first big change, which took place quite a while ago, is related to the form in which information is available. In the past, paper was the most frequently used media for information, and it still is very popular right now. However, more and more information is available through electronic media (mostly on the Internet). Other aspects of information that have changed rapidly in the last few years are the amount of forms that it is available in, the number of sources and the ease with which it can be obtained.

The sheer endlessness of the information available through the Internet, which at first glance looks like its major strength, is at the same time one of its major weaknesses. The amounts of information that are at your disposal are too vast: information that is being sought is (probably) available somewhere, but often only parts of it can be retrieved, or sometimes nothing can be found at all.

This problem has led to the emergence of *Intelligent Agents* that can replace humans for treating information more rapidly and in a more efficient way. In fact, agent research has been going on for about fifteen years before but agents really became a popular word in the computing press around 1994. During this year several key publications appeared. Currently, Intelligent Agent is among the most rapidly growing areas of research and development in computer science.

Originally, the Intelligent Agent is a concept that appeared in Artificial Intelligence (AI) but was quickly taken up by many computer scientists not necessarily working in AI. Indeed, the concept of agent can be adapted to a lot of systems and therefore, there actually exists an important number of research areas that contain the keyword "agent" in their title and that are not related to AI. Finally, we can say that the concept of "Intelligent Agent" is a multi-disciplinary field and regroup several computer domains such as distributed computer systems, software engineering, conception of user interfaces, artificial intelligence, etc. Even if obviously, AI is no more in the centre of preoccupation in agent research, the agent field still offers an opportunity to apply AI techniques to the "real world".

This evolution lead to different conceptions of what an agent is : it can be AI-oriented, software engineering oriented or a balanced mix of these conceptions. Each researcher takes up the concept for his or her own needs and thereby creates his or her own definition of what an agent is. Hopefully, the agent paradigm has today clearly matured and some consensus has been reached to circumscribe the agent's notion.

The success of the agent concept is due to the facilities given by the agent paradigm to develop complex distributed computing systems. Communication between software entities is at the heart of such systems and agents provide neat solutions for communications issues. Agent-oriented techniques have the potential to significantly extend the range of applications that can feasibly be tackled. Analysing, designing and implementing software as a collection of interacting agents may represent a promising point of departure for software engineering to build complex systems. Indeed, applications built upon the agent paradigm already cover a wide number of areas including electronic commerce and information management, health care, process control, etc.

The goal of this dissertation is double. The first goal of this dissertation aims at verifying whether implementing agent-based systems is a good point of departure for software engineering. The first thing to do is to check whether the agent paradigm is actually mature or not. This implies summarising the concepts related to Intelligent Agents and seeing whether enough concepts have been defined to build Intelligent Agents. The second chapter of Part I of our dissertation establishes these theoretical bases.

The second goal is seeing how easy and useful it is to integrate AI techniques in such a system. Can AI make agents behave more intelligently? We introduce in the third chapter of Part I the non-monotonic logic that provides a way of dealing with revisable reasoning.

Afterwards, we must confront these two disciplines to the reality of an application. That is what we do in Part II that focuses on the practical aspects of our dissertation. We present in that part an application we developed. It consists of a small multi-agent system that is endowed with non-monotonic reasoning. Hence, our approach is multi-disciplinary. The application is built with a software engineering view of the agent paradigm but it integrates AI techniques (i.e. non-monotonic logic) and uses distributed object functions (Java methods). The aim here is to examine the requirements of a specific application and analyse how the agent paradigm and the AI provide solutions. This is analysing, designing and implementing an agent-based application using an agent-oriented tool (called Jack) and some AI tools (called Vader, Saten, Hades).

At the end of this dissertation, we should be able to assess the opportunity offered by Intelligent Agents to ease the development of computer systems and to evaluate the place AI can hold in the agent paradigm.

PART I

INTELLIGENT AGENTS AND NON-MONOTONIC LOGIC : AN OVERVIEW

1. Introduction

The areas in which Intelligent Agents technology can be found are getting wider and wider. Intelligent agents are a popular research object these days in such fields as computer science but also psychology or sociology. However, they are most intensely studied in the discipline of distributed systems and Artificial Intelligence. Since many parties use the term "Intelligent Agent" in many different ways, it has become difficult for users to make a good estimation of what the possibilities of the agent technology are. Consequently, a lot has been said about Intelligent Agents but still nobody has come up with a definition on which everyone agrees. In the first section of the chapter 2, we give an overview of some of the existing definitions, based on literature.

In the second section, we single out some properties that can be found in most of the existing definitions. They constitute basic capabilities that every Intelligent Agent *should* possess. We also describe other properties that an agent *could* have. Next, we classify agents into categories according to the properties they possess and we draw up a typology of all the existing agents.

The task of building an agent is not an easy one. Indeed, once the properties the agent should possess are identified, the way to build it (i.e. the architecture) has to be chosen. Whatever the choice, the architecture should provide support for the basic properties of the agent. The fourth section focuses both on the internal and external architectures. External architectures are necessary to describe the external environment of the agent. They are crucial for the standardisation of communication between agents. We continue by explaining the components whose goal is to facilitate the inter-agent communication : the ontology, the content language and the Agent Communication Language (ACL).

Section six focuses on the Mobile Agent technology. Mobile agents are capable of roaming networks for performing a determined task on behalf of its owner and coming back to the user's machine after having completed what the user had asked. This technology takes a large part in the agent research and is very promising. We conclude this first part of our dissertation by providing an overview of some agent-oriented programming languages and of possible application domains.

In chapter 3, we introduce non-monotonic logic because our agent uses this AI technique for reasoning. By doing so, we hope to build a more 'intelligent' agent that is capable of dealing with situations of incomplete or inconsistent information. We chose to endow our agent with Belief Revision and Default Logic capabilities, for they seem to be useful Intelligent Agent reasoning methods. After a brief introduction of what non-monotony is, we talk about the unsuitability of classical logic to formalise non-monotonic reasoning. Afterwards, the second section presents the characteristics of non-monotonic logic. And finally, the third and fourth sections introduce Belief Revision and Default Logic respectively.

2. Intelligent Agents

Since the 1980s, the computer community has been talking about Intelligent Agents but there exists still some difficulty to define the term "agent". In the following chapter, we first discuss different point of views on Intelligent Agents. We then introduce some important concepts related to Intelligent Agents. We conclude this section by taking a look at the language used to build an agent and present some domains of application.

2.1 Definition of an agent

Before we can start talking about agents, it is important to understand the meaning of the concept "agent". Up to now, there is no consensus on the use of a unique definition and the workers involved in agent research have offered a variety of definitions, each hoping to explain his or her use of the word 'agent'. These definitions range from the simple to the lengthy and demanding. Obviously, each of them grew directly out of the set of examples of agents that the definer had in mind.

" ***The MuBot Agent*** *[http://www.crystaliz.com/logicware/mutbot.html] "The term agent is used to represent two orthogonal concepts. The first is the agent's ability for autonomous execution. The second is the agent's ability to perform domain oriented reasoning."*

(...)

The AIMA Agent *[Russel and Norvig 1995, page 33] "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors."*

(...)

The Maes Agent *[Maes 1995, page 108] "Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realise a set of goals or tasks for which they are designed."*

(...)

The KidSim *[Smith, Cypher and Spohrer 1994] "Let us define an agent as a persistent software entity dedicated to a specific purpose. 'Persistent' distinguishes agents from subroutines; agents have their own ideas about how to accomplish tasks, their own agendas. 'Special purpose' distinguishes them from entire multifunction applications; agents are typically much smaller."*

(...)

The Hayes-Roth Agent *[Hayes-Roth 1995] "Intelligent Agents continuously perform three functions: perception of dynamic conditions in the environment; action to affect conditions in the environment; and reasoning to interpret perceptions, solve problems, draw inferences, and determine actions."*

(...)

The IBM Agent *[http://activist.gpl.ibm.com:81/WhitePaper/ptc2.htm] "Intelligent Agents are software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in so doing, employ some knowledge or representation of the user's goals or desires."*

(...)

The Wooldridge-Jennings Agent *[Wooldridge and Jennings 1995, page 2] "... a hardware or (more usually) software-based computer system that enjoys the following properties:*

- *autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;*
- *social ability: agents interact with other agents (and possibly humans) via some kind of agent-communication language;*
- *reactivity: agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the INTERNET, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it;*
- *pro-activeness: agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative."*

(...)

The SodaBot Agent *[Michael Coen http://www.ai.mit.edu/sodabot/slideshow/total/P001.html] "Software agents are programs that engage in dialogs [and] negotiate and coordinate transfer of information."*

(...)

The Brustoloni Agent *[Brustolini 1991, Franklin 1995, p.265] "Autonomous agents are systems capable of autonomous, purposeful action in the real world."* "[1]

Clearly, it is extremely difficult to agree on a definition for the concept 'agent'. The question '*what is an agent?*' is embarrassing for the agent-based computing community in just the same way that the question '*what is intelligence?*' is embarrassing for the mainstream AI community. The problem is that although the term is widely used by many people working in closely related areas, it defies attempts to produce a single universally accepted definition.

There are at least two reasons for this. Firstly, agent researchers do not 'own' this term; it is a term that is widely used in everyday parlance such as in travel agents, estate agents, etc. Secondly, even within the software fraternity, the word 'agent' is really an umbrella term for a heterogeneous body of research and development. The response of some agent researchers to this lack of definition has been to invent yet some more synonyms, and it is arguable that these solve anything or just further add to the confusion. So we now have synonyms including knowbots (i.e. knowledge-based robots), softbots (software robot), taskbots (task-based robots), userbots, robots, Intelligent Agents, personal agents, autonomous agents and personal assistants.

2.2 Properties of an agent

According to us, none of the definitions above can be regarded as "the good definition" but all of them contain some pieces of truth. He notion of agent is very flexible and there is no need to restrict it to one kind of agent. The variety and the flexibility give the agent paradigm its success. Nevertheless, it is useful to single out some agent properties from all these definitions in order to reduce the confusion around the notion of agent. These are properties that make agents different from conventional programs : autonomy, adaptivity and co-operation. Ideally, every agent should exhibit these three primary attributes.

[1] ref. [FRANKLIN & GRAESSER 1996, pp. 1-3]

2.2.1 Autonomy

Autonomy seems to be central to agents. Autonomy refers to the principle that agents can operate on their own without the need for human guidance. Every agent should have a measure of autonomy from its user and act on behalf of this user. Otherwise, it is just a glorified front-end, irrevocably fixed lock-step to the actions of the user. Agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal states and goals (that the agents have to meet on behalf of the user). An autonomous agent can be ***pro-active*** (or *deliberative*) and pursue an agenda independently of its user by acting in a foreseeing goal- or plan-oriented manner rather than acting simply in response to its environment (*reactiveness*). This requires aspects of *periodic action, spontaneous execution,* and *initiative*, in that the agent must be able to take preemptive or independent actions that will eventually benefit the user. The agent does not simply act in response to its environment, it is able to exhibit goal-directed behaviour by taking the initiative.

2.2.2 Adaptivity

The objective of an agent is to enable people to do some tasks better. Since people do not do the same tasks, and since those people who share the same task do it in different ways, an agent must be educable in the task at hand as well as in the way itself to do it. Ideally, there should be components of ***learning*** (so the user does not necessarily have to program the agent explicitly; moreover, certain agents can already learn by 'looking over the user's shoulder) and of ***memory*** (so this education is not lost). The agent has also to be adaptable to changing environment conditions. It has to be ***reactive***, in order to be able to respond to changes of the environment (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined) in reasonable time. A truly useful agent should then be able to adapt its behaviour based on a combination of user feedback and environmental factors.

2.2.3 Co-operation

The user and the agent are essentially collaborating in constructing a contract. The user specifies what actions should be performed on his or her behalf, and the agent specifies what it can do and provides results. This is often best viewed as a two-way conversation, in which each party may ask questions to verify that both sides are in agreement about what is going on. But the agent may also co-operate with other agents to carry out more complex tasks than it can handle itself. Co-operation is just a more specific notion of communication and interaction. It can be seen as a positive interaction.

In order to co-operate, agents need to possess the ability to interact with other agents or humans. For all but the simplest of tasks, we generally need to be assured that the agent shares our agenda and can carry out the task the way we want it done. This generally requires a ***discourse*** with the agent, a two-way feedback, in which both parties make their intentions and abilities known, and mutually agree on something resembling a *contract* about what is to be done, and by whom. This discourse may be in the form of a single conversation, or a higher-level discourse in which the user and the agent repeatedly interact, but where both parties remember previous interactions. We can be more general and mention the notion of ***social ability***. Agents communicate and interact with the user, the system, and other agents via some kind of ***agent-communication language***.

2.2.4 Other properties

Various other attributes are sometimes discussed in the context of agents. For example, ***mobility*** is the ability of an agent to move from one system to another around an electronic network to access remote resources or even to meet other agents. ***Veracity*** is the assumption that an agent will not knowingly communicate false information. ***Introspection*** is the ability to examine and self-reflect its own thoughts, ideas, plan, etc. ***Benevolence*** is the assumption that agents do not have conflicting goals, and that every agent will, therefore, always try to do what is asked. ***Rationality*** is the assumption that an agent will act in a way that is optimal for achieving its goals, and will not act in such a way as to prevent its goals being achieved - at least insofar as its beliefs permit.

2.3 Agent Typology[1]

Every agent does not have all of these properties. Ideally, an agent *should* (but sometimes doesn't) exhibit the three primary attributes : autonomy, adaptivity and co-operation. Beyond this, an agent can be mobile or not, introspective or not, ... We can attempt to place the existing agents into different agent classes thanks to these attributes. We should note that this classification is only one proposal of typology amongst many others. Since there is not one unique definition of what an 'agent' is on which everyone agrees, there cannot exist one right classification of the different types of agent. Therefore, the ensuing list below is in some degree arbitrary, but we believe these types cover most of the existing agent types.

We can classify agents according to the types identified in the schema below, or according to their roles or other attributes they can possess (e.g. mobile, reactive, ...). However, these distinctions are not definitive. For example, with collaborating agents, there is more emphasis on co-operation and autonomy but this does not mean that collaborative agents never learn.

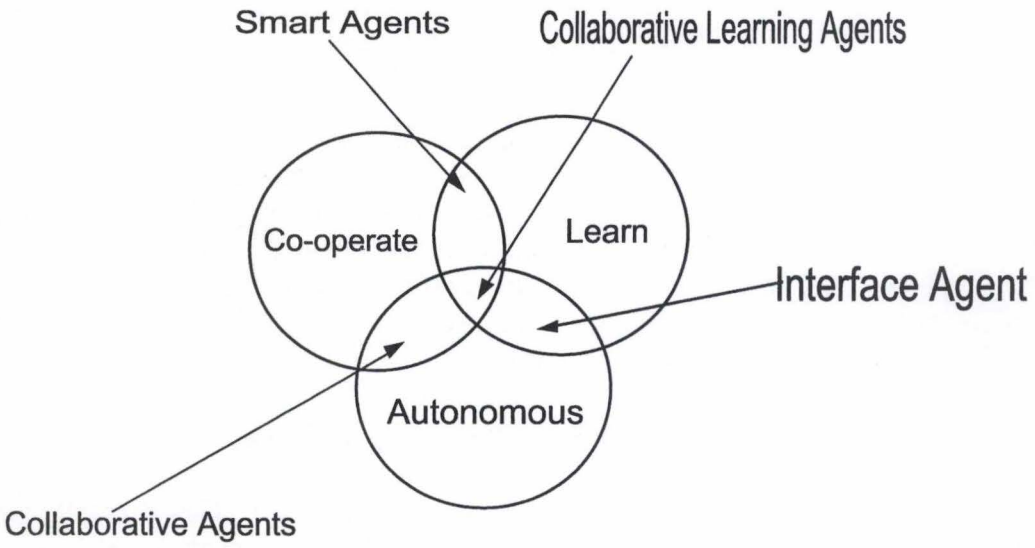


Figure 1 : A partial view of an agent typology[2]

[1] based upon [NWANA 1996]
[2] [NWANA 1996]

We do not consider that anything else that lies outside the intersecting areas are agents. Most expert systems are, for instance, largely autonomous but typically they do not co-operate or learn.

We do not discuss smart agents and collaborative learning agents. Smart agents are autonomous agents capable of learning and collaborating. Moreover, they are an aspiration of agent researchers rather than reality. Truly smart agents do not yet exist.
We identify six types of agents :

2.3.1 Collaborative agents

Collaborative agents emphasise autonomy and co-operation (thanks to an agent-communication language) in order to perform tasks for their owners. They may learn, but this aspect is not a major emphasis of their operation. Some AI researchers are providing stronger definitions for such agents, e.g. some attribute mentalistic notions are hereby used, such as Belief, Desire and Intention (BDI) type agents (we look at BDI more into detail in 2.4.1).

The key properties of these agents are autonomy, social ability and pro-activeness. Hence, they should be able to act rationally and autonomously in open and time-constrainted multi-agent environments. Most currently implemented collaborative agents do not perform any complex learning, though they may or may not perform limited parametric or rote learning.

2.3.2 Interface agents

Interface agents emphasise autonomy and learning in order to perform tasks for their owners. They are like personal assistants who are collaborating with the user (not with other agents as is the case with collaborative agents) in the same work environment. An explicit agent-communication language is not always required for that kind of collaboration.

Generally, interface agents support and provide assistance, typically to a user, learning to use an application or an operating system. They 'watch over the shoulder of the user', make suggestions to perform some tasks better, etc. They learn to better assist the user by observing and imitating him/her, by receiving positive and negative feedback from him/her, by receiving explicit instructions from him/her or sometimes by asking other agents for advice.

2.3.3 Mobile agents

Mobile agents are agents capable of roaming networks (such as Internet), interacting with foreign hosts, gathering information on behalf of its owner and coming back to the user's machine having performed what the user had asked (e.g. managing a telecommunications network, reserving a flight, ...). They are autonomous and they co-operate (but in a different way as collaborative agents do). In fact, the general perception of agents is often synonymous with mobile agent (we go deeper into the subject in section 2.6).

2.3.4 Information/Internet agents

Information agents are tools to help the user manage the explosive growth of information he/she has to face in these turbulent times. They perform the role of managing, manipulating or collating information from many distributed sources.

There exists a subtle distinction between information agents and some 'information-specific' interface agents or collaborative agents (e.g. whose duty is filtering a stream of data). Interface or collaborative agents started out quite distinctly, but with the explosion of the Internet (and their applicability to it), there is now a significant degree of overlap. This is inevitable, especially since information agents are defined using different criteria. They are defined by *what they do*, in contrast to collaborative or interface agents who are defined -via their attributes- by *what they are*. Interface and collaborative agents can, in a sense, be information agents if they are employed in Internet-based roles.

2.3.5 Reactive agents

Reactive agents represent a special category of agents that do no possess internal, symbolic models of their environment. They act reactively, i.e. in a stimulus-response manner to the current state of their environment. Hence, there are no a priori specifications (or plans) of the behaviour in their set-up.

Many theories, architectures and languages exist for this type of agents. But most interestingly, these kind of agents are relatively simple and interact with other agents in basic ways (however, all the interacting agents viewed globally can show some complex patterns of behaviour). A reactive agent can be seen as a collection of modules that operate autonomously and are responsible for specific tasks. Communication between the modules is minimised and is of relatively low-level nature.

Reactive agents tend to operate on representations that are close to raw sensor data, in contrast to the high-level symbolic representations that are abound in the other types of agents discussed so far.

2.3.6 Hybrid agents

Since each type of agent has its own strengths and deficiencies, the trick is to maximise the strengths and minimise the deficiencies. One way of doing this is to adopt a *hybrid* approach, by bringing some of the strengths of different paradigms together. Therefore, hybrid agents are agents whose constitution is a combination of different agent philosophies within a singular agent.

2.4 Agent Architecture[1]

With the previous sections, we perceive better what an agent is. In this section, we can thus analyse how they are built. We will discuss two levels of architecture. The first level of architecture defines the relationships between the components that constitute the internal composition of an agent; we will call this the agent internal architecture. Secondly, we will take a look at the architecture of the Multi-Agent Systems (MAS), which defines the relationships and interactions between each of the individual agents. The Multi-Agent Systems level architecture often can be seen as the co-ordination mechanism to which the total agent system should conform.

[1] based upon [FLORES-MENDES 2000], [BUSETTA & KOTAGIRI 1998], [HAYZELDEN & BIGHAM 1998], [OMG 2000]

2.4.1 Agent internal architecture

There are many ways to build an agent and there are as many architectures. In any case, the agent should provide support for the basic properties of an agent. For example, an agent's behaviour includes autonomy, reactivity, pro-activeness, social ability (co-operation and communication), learning, goal orientation, mobility and so on. A particular agent might not have all of these properties but might still be considered as an agent.

An approach that has received a great deal of attention is the so-called Belief, Desire, Intention, (BDI) architecture. The BDI architecture is based on the study of the mental attitudes.

The **beliefs** represent the informational state of a BDI agent, what it knows about itself and the world. **Desires** or goals are its motivational state, that is, what the agent is trying to achieve. The **intentions** represent the deliberative state of the agent, that is, which plans the agent has chosen for eventual execution. The agent reacts to events that are generated by modifications to its beliefs, additions of new goals or messages coming from the external world. Intentions are executed one step at the time. A step can query or change the beliefs, perform actions, suspend the execution until a certain condition is met, and submit new goals. The operations performed by a step may generate new events which, in turn, may start new intentions.

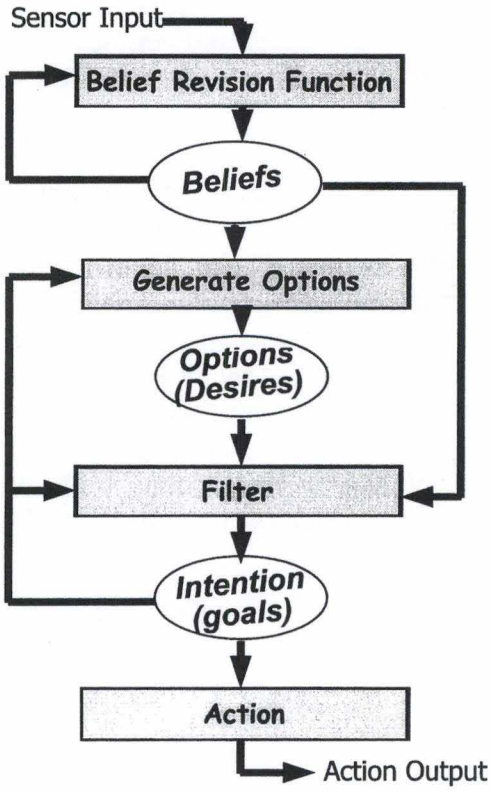


Figure 2 : Belief-Desire-Intention architecture[1]

[1] from [WOOLDRIDGE 1999]

Based on previous research and practical applications, Rao and Georgeff[1] have described a computational model for a generic software system implementing a BDI agent. Such a system is an example of event-driven programming. In reaction to an event, for instance a change in the environment or its own beliefs, a BDI agent adopts an appropriate plan.

Figure 2 shows a BDI architecture. A BDI agent performs four functions : A Belief Revision function that generates a new set of beliefs, given some sensor input ; an option generation function that determines the options available to the agent, given the current beliefs and current intentions ; a filter that generates a new set of intentions given current beliefs, options, and intentions ; an action selection function that determines an action to perform, given current intentions.

It has been shown that BDI is well suited to modelling certain types of behaviour, such as an application of standard operational procedures by trained staff. It has been successfully adopted in fields as diverse as simulation of military tactics, application of business rules in workflow, and diagnostics in telecommunications networks.

In summary, BDI is the abstract architecture of a family that allows a high degree of sensitivity to the context when deciding how to react to changing conditions.

2.4.2 Multi-Agent System (MAS) architecture

The previous section treats the internal architecture of an agent. Since agents act in an environment that contains other agents and various services, an architecture should also describe the external environment. Such architectures standardise communication between agents and access to common services.

There are two approaches to build a MAS. Agent system architectures can be language and platform independent, or they can be based on a particular language or implementation framework. There are numerous language specific frameworks for implementing agents. The problem with this approach is that it can be difficult, or even impossible, for agents implemented using different frameworks to communicate. The language specific approach can be useful for implementing custom agent systems, for instance, within a company. For more general applications, a platform independent architecture is needed.

Recently, several independent research groups started to pursue the standardisation of MAS. Two of them are the Object Management Group (OMG) and the Foundation for Physical Agent (FIPA).

[1] see also [RAO & GEORGEFF 1995]

FIPA[1]

FIPA is a standardisation approach for a complete architecture for supporting Intelligent Agents. The FIPA architecture consists of the following concepts :

- **Agent** : Each agent has an identifier that is unique in the FIPA agent universe.
- **Agent platform (AP)** : A complete agent system consisting of a directory agent, a management agent and a communication channel agent.
- **Directory Facilitator (DF)** : The directory is an agent that provides catalogue services to the other agents. The agent defines an agent domain and supports the following actions : register, deregister, search, and modify.
- **Agent management System (AMS)** : This is an agent that manages the activities within an agent platform, including creation of agents, deletion of agents, and migration of an agent to and from other platforms. It supports the actions authenticate, register-agent, deregister-agent, and modify-agent.
- **Agent Communication Channel (ACC)** : The communication language used in the FIPA architecture is based on the speech act theory, where messages are viewed as communicative acts intended to perform an action. Communication between agents necessitates ontologies, a knowledge representation language (KIF) and a communication language similar to KQML (Knowledge Query and Manipulation Language; see section 2.5 for more information about Agent Communication Languages).

FIPA has demonstrated several applications implemented using their architecture and it seems as if FIPA is an accepted standard for agents.

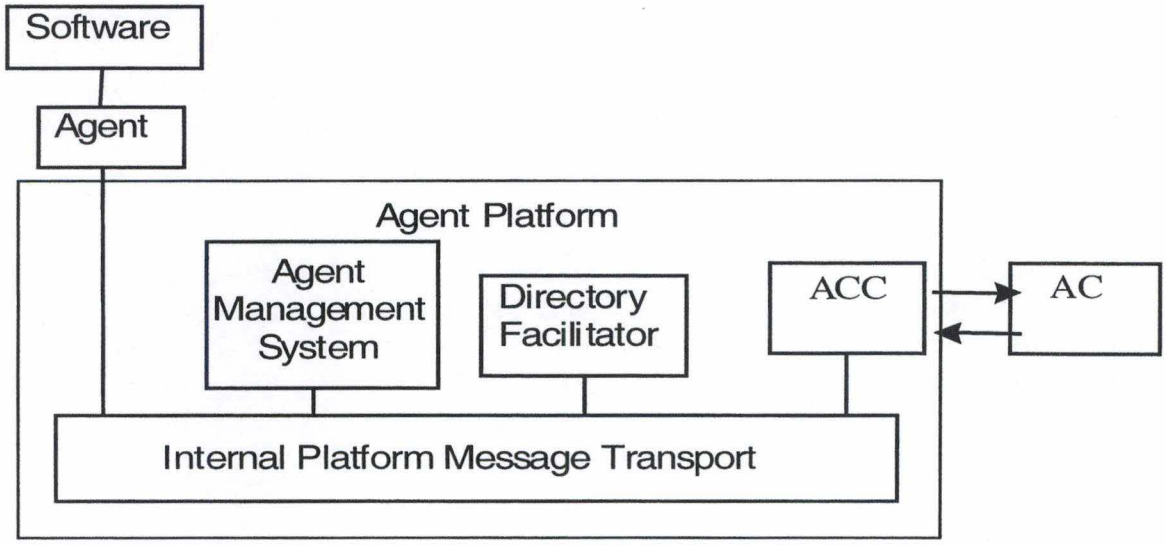


Figure 3 : FIPA-Agent Management Model[2]

[1] http://www.fipa.org/
[2] from [O'BRIEN 1998]

OMG[1]

In 1989, a consortium of object vendors grouped together to form the Object Management Group (OMG). Since then, they have been defining standards and architectures that allow object components written by different vendors to inter-operate across networks and operating systems. The specification of the architecture is referred to as the Common Object Request Broker Architecture (CORBA). Because of some similarities between the concepts of object and agent, the OMG has created an Agent Working Group with the mission to provide a forum for identifying and building consensus and convergence in the industry around agent technology.

Because it is likely that agent technology and object technology specifications will eventually overlap, it is desirable not to duplicate the specifications. Indeed, both technologies need namespaces, have lifecycle services, use persistence, address mobility, etc.

An OMG-FIPA Liaison-Project[2] was proposed to encourage agent technology standards to evolve consistently with object technology standards and to further co-ordination between OMG and FIPA's related work toward an agent technology standard.

2.5 Agent communication components

Agent-to-agent communication is key to realising the potential of the agent paradigm. Indeed, we have seen that one of the agent's properties is co-operation. Most people agree that this implies that agents have to communicate and interact with other agents.

However, some say that agents could work together in multiple agent systems (MAS) as a perfectly co-ordinated team without having to communicate. This would be possible if every agent had perfect information about the state of the entire system - if every agent knew what every agent knew, what every agent intended to do, and how everything in the system stood in relation to everything else in the system. Of course, such perfect knowledge is often impossible. Indeed, agents have at best partial, possibly incorrect information about the state of their environment. This is why the agents have to communicate to co-operate.

Agents can use three elements to facilitate inter-agent communication. The first element is an ***ontology*** that is used to understand the semantic of a message content. The second element is a ***content language*** that is used to formalise the knowledge and information. It defines in a way the syntax of the message content. The third element is an agent-communication language. Agents use ***an Agent Communication Language*** or ACL to transmit information and knowledge. For example, an ACL specifies the type of message or the receiver of the message. We will discuss these three elements below.

[1] http://www.omg.org/
[2] see also [MCCABE 1993],

2.5.1 Agent Communication Languages[1]

Where agents need to communicate, they must individually understand some agent communication languages (ACL). An ACL should allow agents to enlist the support of others to achieve goals, to monitor their execution, to report progress, success, failure, to acknowledge receipt of messages, to refuse task allocations and to commit to performing tasks for other agents.

There are two main approaches to design an agent communication language. The first approach is procedural, where communication is based on executable content. This approach allows programs to transmit not only individual commands but also entire programs. The second approach is declarative, where communication is based on declarative statements, such as definitions or assumptions.

Proprietary ACLs have been chosen as the communication mechanism by most researchers. This is probably due to their application requirements being specific and due to the lack of standardised ACL being available. A standardised ACL is essential if agent based network management systems are going to achieve their integration into legacy systems and inter-operate with future demands for network system upgrades.

One of the more popular agent-languages is the Knowledge Query and Manipulation Language (KQML). KQML is an evolving standard ACL, being developed as part of the DARPA Knowledge Sharing Effort (KSE). KQML is a communication language and protocol for exchanging information and knowledge and most declarative language implementations are based on illocutionary acts, such as requesting or commanding. Such actions are commonly called "performatives". At the heart of KQML are more than three dozen performatives that define the allowed speech acts agents may use. The table below presents the KQML performatives.

CATEGORY	RESERVED PERFORMATIVE NAME
Basic informational performatives	Tell, deny, untell, cancel
Basic query performatives	Evaluate, reply, ask-if, ask-about, as-one, ask-all, sorry
Multi-response query performatives	Stream-about, stream-all
Basic effector performatives	Achieve, unachieve
Generator performatives	Standby, ready, next, rest, discard, generator
Capability definition performatives	Advertise
Notification performatives	Subscribe, monitor
Networking performatives	Register, unregister, forward, broadcast, pipe, break
Facilitation performatives	Broker-one, broker-all, recommend-one, recommend-all, recruit-one, recruit-all

[1] based upon [NWANA & WOOLDRIDGE 1996], [GENESERETH & KETCHPEL 1994], [FININ & LABROU 1999], [LABROU et al. 1999]

The KQML language can be viewed as consisting of three layers : the **content**, **message** and **communication** layers.

The **content** layer specifies the actual content of the message. This layer contains an expression written in a language allowing to encode the information. In this context, it is important to agree on the use of a common language such as KIF (Knowledge Interchange Format, see 2.5.2) to support the knowledge sharing. The KQML standard itself has nothing to say about this. The set of performatives constitutes the **message** layer. The **communication** layer encodes low level communication parameters, such as the identity of the sender and the recipient.

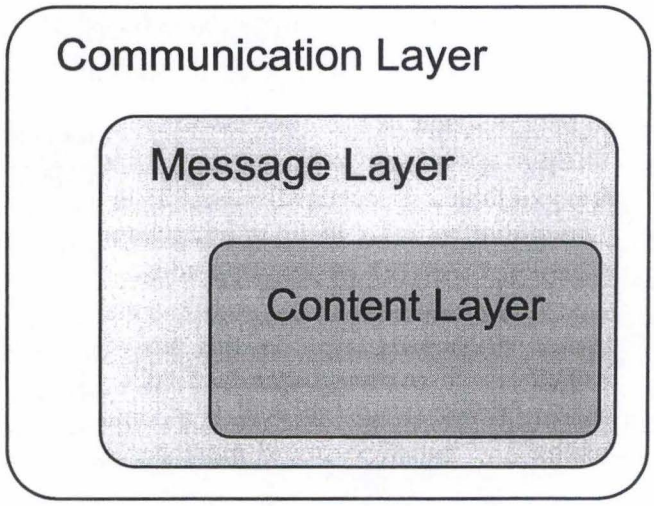


Figure 4 : An abstract view of the KQML language

The Foundation for Intelligent Physical Agents (FIPA) has also developed an ACL as well. Like KQML, FIPA's agent communication language is also based on speech act theory : messages are actions or communication acts. The FIPA ACL specification consists of a set of message types and the description of their pragmatics - that is, the effects on the mental attitudes of the sender and the receiver agents. FIPA ACL is superficially similar to KQML except for different names for some reserved primitives. Thus, it maintains the KQML approach of separating the outer language from the inner language. The two languages differ in the details of their semantics framework. This makes it impossible to come up with exact mappings or transformations between KQML performatives and their completely equivalent FIPA primitives, or vice versa.

Examples of KQML messages[1]

In the following example, the KQML performative is *tell*. The agent that is sending the message seeks to inform a customer, customer-2, concerning a quote for performing a service, service-4, in reply to an earlier request from customer-2, to BT customer services. The content of the message is expressed in standard Prolog and the ontology for BT's services domain is assumed.

```
(tell
      : content  "cost(bt,service-4, £5677)"
      : language standard prolog
      : ontology       bt-services-domain
      : in-reply-to    quote service-4
      : receiver customer-2
      : sender   bt-customer-services)
```

In the next example, the message is expressed in KIF. The content of this message says that the torque of object motor1 at simulation time (sim-time) 5 is 12 kg. It is assumed that the ontology *'motors'* defines the terms torque, sim-time, kgf, and so on.

```
 (untell
      : language KIF
      : ontology motors
      : in-reply-to    S1
      : content  (=  (val  (torque  motor1)  (sim-time  5))  (scalar
12kgf)) )
```

The concept of a standard communication language for software agents that is based on speech acts has found wide appeal. KQML was among the first such ACLs to be developed and used. Nevertheless, after eight years of experimentation, there are still signs of immaturity. In general, different KQML implementations cannot inter-operate (but this is mainly due to a lack of motivation). However, KQML has played an important role in defining what an ACL is and what the issues are.

2.5.2 Content languages[2]

As we have said in the previous section, the KQML standard says nothing about the content layer. The role of the content language is to provide an interlingua for a wide range of systems. A variety of content languages have been used with ACL's, including Prolog, SQL, etc. However, the best known work in this area is the ARPA Knowledge-Sharing Effort (KSE). One of the results is the Knowledge Interchange Format (KIF) that provides an interlingua for knowledge bases to inter-operate.

---

[1] from [GENESERETH & KETCHPEL 1994]

[2] based upon [NWANA & WOOLDRIDGE 1996], [GENESERETH & KETCHPEL 1994]

In other words, as shown on Figure 5, for two agents with different legacy knowledge bases to inter-operate, both knowledge bases could be translated into KIF that will be the shared representation language. KIF provides also a common language for reusable knowledge. It is possible to build library containing knowledge bases written in KIF.

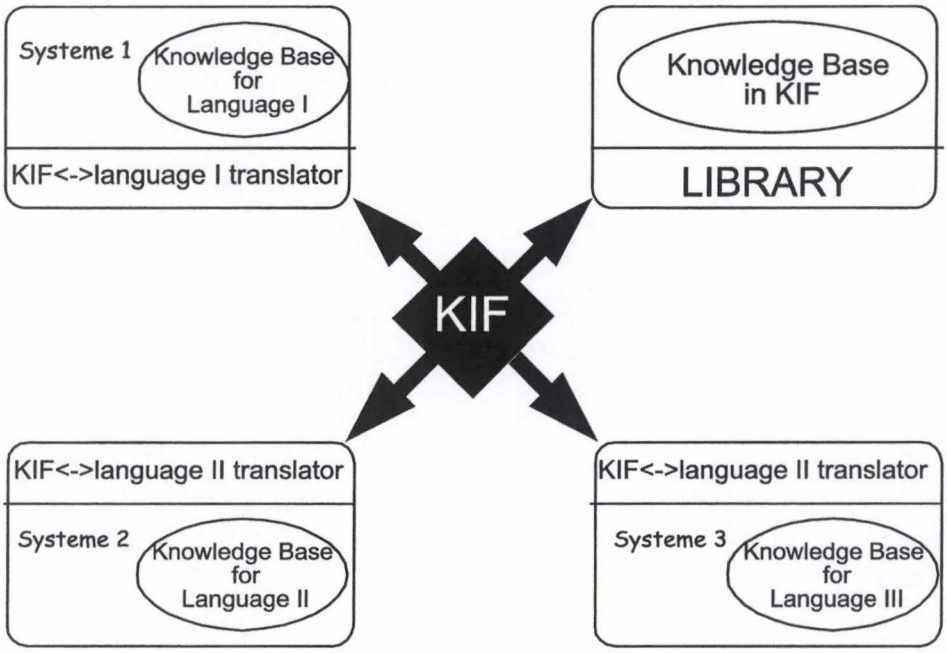


Figure 5 : Knowledge Interchange Format[1]

KIF is a prefix version of first-order predicate calculus, with various extensions to enhance its expressiveness. It provides for the encoding of simple data, constraints, negations, disjunctions, rules, quantified expressions and so forth. KIF also includes an axiomatic specification of large function and relation vocabulary and a vocabulary for numbers, sets, and lists.

Some KIF software products are available. For example, partial translators exist for a number of other Knowledge Representation (KR) languages, such as Prolog. Parsers for KIF also exist which transform KIF strings into C++ or Java objects.

The future of KIF outside the AI-related community remains unclear because it may not be acceptable to a wider community since it's too logic-oriented. Nevertheless, its expressive power may be decisive. KIF is up to now the only widely used interlingua for Knowledge Base systems.

[1] from [FININ & LABROU 1999]
[2] from [GENESERETH & KETCHPEL 1994]

Examples of KIF expression[1]

KIF provides for the expression of simple data. For example, the sentences shown below encode 3 tuples in a database. The first argument in each is the social security number of an individual, the second argument is the department within wich the individual works, and the third argument is the individual's salary.

```
(salary 019-46-9532 widgets 72000)
(salary 021-45-3179 grommets 36000)
```

More complicated pieces of information can be expressed through the use of complex terms. For example, the following sentence says that one chip is larger than another

```
(> (*(width chip1) (length chip1)) (* (width chip2) (length chip2)))
```

2.5.3 Ontology[1]

An important pre requisite to communicate is using the same concepts. ***Ontology*** treats this issue. In philosophy, it refers to the subject of existence. In computer sciences, the term ontology is used to indicate a specification of a conceptualisation. An ontology is a description of the concepts and relationships that can exist for a community of agents. Practically, an ontology is a specification used for making ontological commitments and an ontological commitment is an agreement to use a shared vocabulary in a coherent manner. The concept of ontology is therefore very important in the context of knowledge sharing. Indeed, without agreement on a particular ontology between the agents, it is impossible to share information and thus to co-operate. An agent commits to an ontology so that he/she can communicate about a domain of discourse with other agents. Pragmatically, a common ontology defines the vocabulary with which queries and assertions are exchanged among agents. Creating an ontology involves explicitly defining every concept to be represented. For example, to plan a trip, we need to be aware of ontology concepts such as planes, flights, airports...

Three important aspects to explicit ontologies are the conceptualisation, the vocabulary and the axiomatisation. Conceptualisation involves the underlying model of the domain in terms of objects, attributes and relations. Vocabulary involves assigning symbols or terms to refer to those objects, attributes and relations. Axiomatisation involves encoding rules and constraints that capture significant aspects of the domain model. Consequently, two ontologies may differ from one another for three different reasons. First they may be based on different conceptualisations. Secondly, they may be based on the same conceptualisation but use different vocabularies. Finally, they may differ on how much they attempt to axiomatise the ontologies.

[1] based upon [GRUBER 1993], [NWANA & NDUMU 1999], [FAQUAR et al.]

Often, applications define their own limited ontologies for their limited applications. These ontologies are generally implicit. Thus, the ontology of two different systems would certainly be different. This is why there are currently efforts to develop explicit ontologies that can be shared across disparate software developers. Declarative representation in a well-defined knowledge representation language can be used to build an explicit ontology.

As mentioned in the previous section, **KIF** can be used to represent reusable knowledge. Therefore, it can also be used to represent declarative ontologies.

Ontolingua[1] is another ARPA-sponsored effort towards reusable ontologies. Ontolingua is a language for building, publishing, and sharing ontologies. Ontolingua statements and axioms are written in an extended KIF notation and natural language sentence. An "ontolingua server" has been developed at the University of Stanford. The tool uses an extended ontolingua language. It makes use of the World Wide Web to enable wide access and provide users with the ability to publish, browse, create, and edit ontologies stored on the ontology server. Ontology construction is difficult and time consuming. With the ontolingua server, users can quickly assemble a new ontology from a library of modules.

There are also several efforts aimed at defining and generating large ontologies. These are ontologies that are not directed at specific domains. A good example of such an ontology is WordNet. It is an on-line lexical database with more than 1.666.000 word forms. WordNet includes many semantic relationships between words and word senses, and it also contains several semantic relations including synonymy, antonymy, hyponymy, meronymy and troponymy.

Works are also made in specific domains. One example is the Unified Medical Language System. The purpose of the UMLS is to aid the development of systems that help health professionals and researchers to retrieve and integrate electronic biomedical information from a variety of sources. The goal is to make it easy for users to link disparate information systems, including computer-based patient records, bibliographic databases, factual databases, and expert systems. The UMLS is a complex collection of medical terms and relationships derived from standard classifications.

However, WordNet and UMLS are not really designed for agent applications but it is possible to use them to complement further the ontology defined for some agent applications. These efforts show that the ontology problem is not reserved for agent systems.

Anyway, it is proven that reusable ontologies are becoming increasingly important for tasks such as information integration and knowledge development.

2.6 Mobile agents[2]

In section 2.3.3, we have seen a special kind of agents : the mobile agents. We can not close this chapter without speaking a little bit more about the mobile agents. Indeed, this technology takes a large part in the agent research and promises a lot. A sign of this is that the public perception of agents is sometimes synonym with mobile agents.

---

[1] see http://ontolinguia.stanford.edu

[2] based upon [HAZELDEN & BIGHAM 1998], [BERNEY]

As we have already mentioned, mobile agents are agents capable of roaming networks, interacting with foreign hosts, gathering information on behalf of their owners and coming back to the user's machine having performed what the user had asked. Mobile agent technology was originally concerned with the ability to move executable code from one computer to another. The main benefit from adopting this approach was that available computational resources on another computer could be utilised. Researchers became then interested in dealing with the idea of moving so called 'intelligence' from one place of execution to the next.

Using mobile agent technologies can provide some advantages. One of them is that this technology can reduce network resource utilisation because there is no need to maintain active processes at two places. However, it still remains the case that passing of messages is often more efficient than moving an agent from one place to another and, therefore, careful analysis of the benefits that each approach can provide in a particular application domain, needs to be made. For agent gathering information, and filtering out unwanted information, it makes sense for them to be operating on the machine where the data is located.

A major concern with mobile agent technology is the lack of security on the agent side relationship. For example, there is considerable concern about users delegating confidential information (such as credit card details) to their mobile agent and releasing them into an open network system, where, without proper security considerations, server computers could gain access to confidential information. There are also concerns on the server side relationship. Many companies erect firewalls to protect their data from external access of mobile agent entering the company network. In theory, the mobile agent should not be able to have access to a host machine and to data structures. Nevertheless, there is always the possibility of malicious agents containing hostile code.

Another concern about mobile agents is platform incompatibility. Each platform must support the code format of the agents. An alternative for the code format is virtual machine codes, like Java byte code. This is the main reason of the success of Java. It is widely accepted because it provides platform independence for mobile agents. However, it can be more difficult and complex in some cases to support than an Agent Communication Language (ACL) for static agents.

2.7 Agent implementation languages[1]

2.7.1 Characteristics of agent implementation languages

To implement an internal agent-architecture and a MAS-architecture, appropriate languages are needed. At first glance, it seems that every mature language can be used to develop an Intelligent Agent. However, the language has to provide some tools to build the agent's characteristics. Two important functionalities of an agent are communication and knowledge representation. The knowledge has to be represented in such a way that the communicating peers understand each other. Advanced agents require some kind of knowledge representation and processing methods (i.e. for describing ontologies and rules and for doing some reasoning using an inference machine).

[1] based upon [FININ & LABROU 1999], [VERSTEEG & STERLING 1997]

Traditional languages are still used to construct agent applications. Typically, object-oriented languages such as Java or C++ lend themselves more easily for the construction of agent systems. This is because the concept of an agent is not too distant from the concept of object and especially from the concept of distributed object. Indeed, distributed objects are appropriate to build decentralised systems and agent-based systems are also decentralised systems. Thus, it seems to be natural to build agents with such languages. Moreover, agent-based languages are often inspired from object-based languages, like Java.

In many cases, it is easier to use an agent specific language. Two approaches are necessary to have a good agent language : an external behaviour-oriented approach and an internal behaviour-oriented approach. A combination of the two approaches yields a good language.

The first approach, based on the distributed computing language, focuses on the communication abilities of the agents. This type of languages provides strong support for security and information passing. Communication support is a "must". There should be a very easy way to describe information exchange between agents.

The second approach, based on languages coming from the Artificial Intelligence, provides extended support for knowledge representation and reasoning. Although the behaviour of an agent can be described using an imperative language, a natural way is to have a rule-based declarative language. Declarative or functional languages are sometimes more appropriate.

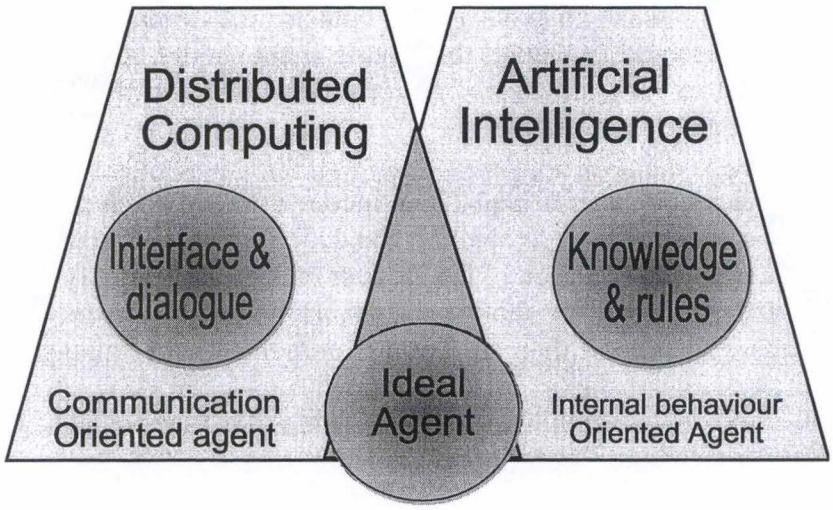


Figure 6 : Two approaches : Distributed Computing and Artificial Intelligence[1]

A lot of languages and frameworks designed specially for developing intelligent software agents already exist. We can distinguish between two classes of agent implementation languages : the new programming languages usually based on old languages and add-ons which offer new libraries or classes for old languages.

There is no "best" language for implementing agents or agent systems. The "best" language is the one (or are the ones) which suits best for the application to be developed. The current approaches are emerging from distributed computing and from Artificial Intelligence. Better and better languages and frameworks will eventually appear, borrowing the best attributes from both sides.

[1] from [FALKENROTH & GRANLUND 1998]

In part II, we will describe in detail an agent oriented language called Jack. One important characteristic of Jack is that it is a Java-based language.

2.7.2 Mobile agent language

Mobile agents have some specific characteristics and they need therefore specific. One of the characteristics is that mobile agent programs are only able to run on hosts that have an execution environment that interprets the language they were written in. It is possible for an agent server to be able to support more than one language, however there are presently many competing and incompatible types of agent servers, each only capable of interpreting at most a few languages.

Therefore, in theory, the only necessary condition of a mobile agent language is that an execution environment on the host supports the language. However, in reality, other conditions are required for such a language. The language must be able to support agent migration, communication between agents, access to server resources, security mechanisms, appropriate efficiency and the ability to run on multiple platforms.

Some languages such as Obliq and Telescript have been specifically designed for writing mobile agents. There are also many mobile agents being written in general purpose languages extended with a special library. Below is a brief description of some of the languages that have been used to write mobile agents.

Telescript is a proprietary system developed by General Magic. The Telescript language has been specifically designed for implementing mobile agents. Telescript is a complete object oriented language with a syntax in many ways similar to that of C++. Telescript has a library of built-in classes for writing mobile agents. The Telescript language has commands for agent migration and inter-agent communication. The Telescript language has had a great influence on the development of mobile agents and mobile agent language.

Java is a general purpose language. It is an object-oriented language. While Java was not specifically designed for writing mobile agents, it has most of the necessary capabilities for mobile agent programming. Java is multi-threaded. Java programs are able to run on any platform with a Java Virtual Machine interpreter. The Java libraries provide good support for communication procedures. As a consequence, Java has been used as the basis for many implementations of mobile agent systems.

One of the Java-based systems is the IBM's **Aglets** under development by the IBM Research Centre, in Japan. An Aglet is a mobile agent that is derived from an abstract class called Aglet. Aglets use an event-driven approach to mobile agents. Each aglet implements a set of event handler methods that define the aglet's behaviour.

Another system based on Java is **Odyssey** developed by General Magic. The attempt is to achieve the functionality of Telescript, using Java.

Obliq is an experimental language under development by the Digital Equipment Corporation's Systems Research Centre. Obliq is a lexically scoped, object-based, interpreted language that supports distributed computation. Obliq has built-in procedures for importing and exporting procedures and objects between machines. Obliq's semantics of network

computing is fundamentally different from the other languages considered. Whereas other languages see each computer as independent worlds that can communicate with each other through networks, Obliq treats the network as a single computer with sites as components.

AgentTcl is a mobile agent system being developed by Dartmouth College. The AgentTcl language is an extension of the Tool Command Language (Tcl), the language extensions add commands for agent migration and message passing. The extra commands give AgentTcl scripts mobility capabilities similar to Telescript.

There are currently many competing languages. Telescript is one of the best languages for implementing mobile agents. But the problem with Telescript is that it is a proprietary software and that it is a closed standard. Java, on the contrary, is an open specification. An open standard system that delivers the same functionalities to the user can be expected to gain a greater market share than a proprietary technology. What is certain is the fact that only a few languages will gain enough support to enable the vision of mobile agents roaming the Internet to become reality.

2.8 Applications of Intelligent Agents[1]

Now that we have drawn up an overview of the agent paradigm, we have a look here at the way the agent techniques are used in real applications.

Agents can be a useful tool due to the exponential growth of information available on the Internet. With individuals adding information on the Internet regularly, the influx of data is difficult to process. Users are beginning to suffer from severe information overload. This problem is compounded by the fact that speed of access to the Internet is also increasing, particularly with the development of technologies such as ISDN. With this new technology, many users will acquire the knowledge they seek in a matter of minutes, yet for the thousands of users who have yet to upgrade to faster pathways, agents are the smartest way to obtain valued information.

Some agents, which are available now, can save users time by performing repetitive tasks such as gathering and posting e-mail and checking newsgroups. This type of agents are relatively simple and are often rule-based scripts. Agents that search for information will extend this advantage of walking away from the computer and having time to do other things while your agent searches. Currently, if you need some information, you use a search engine such as Yahoo or AltaVista, but this can take you minutes, hours, maybe even days to search and gather information. With the help of an Intelligent Agent, this time could decrease to a few minutes. You tell the agent what you want and it will do the job for you.

Moreover, agents will make the Internet and computers easier to use. The rapid progression of technology has caused the complexity of hardware and software to increase. At the same time, the decreasing cost of agent technology is allowing access to more people with fewer computer skills.

The current applications of Intelligent Agents are of a rather experimental and ad hoc nature. Besides universities and research centres, a considerable number of companies are doing research in the area of agents. To make sure their research projects will receive further

[1] based upon [DO et al.], [BLOCH & SEGEV 1996]

financing, many researchers are nowadays focusing on rather basic agent applications, for these lead to demonstrable results within a definite time.

The current trend in agent developments is to develop modest, low-level applications. Yet, more advanced and complicated applications are more and more being developed as well. At this moment, research is mostly being done into separate agents, such as mail agents, news agents and search agents. This is the first step towards more integrated applications, where these single, basic agents are used as the building blocks. Combination of basic agents creates complex structures that are able to perform high-level tasks for users, suppliers and intermediaries. The interface to this system is through a single agent which delegates sub-tasks and queries to other agents.

The most popular application of Intelligent Agent technology is electronic commerce. But other interesting applications are arising : messaging, adaptive user interfaces, systems and network management, administrative management, collaboration, mobile access and information management.

2.8.1 Electronic Commerce

The Internet is becoming an increasingly important channel for retail commerce as well as for business-to-business transactions. For sheer convenience and preservation of time, consumers are looking for suppliers selling products and services on the Internet. Meanwhile, suppliers are looking for buyers to increase market share. Both consumers and buyers need to automate handling of their "electronic financial affairs".

But electronic commerce has to deal with two problems : the too vast amount of information on the Internet and the lack of a feasible electronic monetary system. This has led to the relatively slow growth of electronic commerce.

People have speculated that Intelligent Agents will be able to solve these problems. Intelligent Agents are able to sort through the clutter on the Internet, resulting in the selection of specific brands, products, and stores. These entities will also be able to speed up the process of locating items on the Internet and leave users more time to do other, more productive and enjoyable tasks.

Intelligent Agents can assist in electronic commerce in a number of ways. Agents can "go shopping" for a user, taking specifications and returning with recommendations of purchases which meet those specifications. They can act as "salespeople" for sellers by providing product or service sales advice, and they can help troubleshoot customer problems.

Perspectives for buyers and sellers

The Internet is causing a problem for the buyers. The consumer is not able to examine objects or make comparisons between different products as easily on the Internet when paralleled with conventional shopping. Products are not placed side by side on a shelf to compare quality, nor are the sellers located near each other to compare prices. Intelligent Agents can go searching for products for a consumer by using things such as store locators, brand locators, category locators, and product locators. They will also be able to query the user's opinions on certain products. In short, the consumer will need to make certain specifications and the agent will then find products that match these specifications.

The Internet is also causing a problem for the sellers. Buyers cannot be found, and this is the problem for marketers on the Internet. Intelligent Agents can help to solve this problem by acting as a virtual salesperson for the company. Sellers will also be able to create representative agents to provide expert advice, thus helping to address common customer problems.

Filter of information

Consumers are confronted with information overload when they go shopping on the Internet. Consumers want to be able to find a broad selection when looking for a product. However, the selection must be edited down to reasonable and manageable proportions. This editing feature is the job of the Intelligent Agent. Without this editing feature, buying a simple music CD would be quite an experience: imagine walking into a mall and finding 200 different music stores!

Decision agent and demand agent

Marketing on the Internet will be affected by two types of agents, demand agents and decision agents. Demand agents work in the interest of the provider. These agents are trained with product knowledge. They represent the products and services and transfer that information to the decision agent. Decision agents work for the consumers. The consumer trains them to search for products and services that are represented by the demand agents. They make recommendations based on the preferences of the consumer. The demand and decision agents work together. The decision agents "meet" with the demand agents to gather information requested by the user.

The future

Intelligent Agents may cause a shift in consumer loyalty. With the increased use of Intelligent Agents, people will become more brand loyal than store loyal. The store will lose some of its identity, because it is not a physical environment in which a consumer enters and spends time.

Another big change is that providers will be able to watch their products sold in real-time. There will not be a long turn-around time to see the effects of advertising and marketing. Instead of having to market in a conventional way, businesses will see results much faster. This also means that they will have to react to changes in an accelerated manner. This could force a company to continually change and update their products and operations.

Consumers will not have to fumble through lists and lists of sites to make purchasing decisions. Instead they will ask their agent to start searching, walk away, and come back to find the information they want. Ultimately, consumers will have their own personally trained shopper and research assistant, who knows all of their preferences, goals, and information desires.

2.8.2 Messaging

Messaging software is also an area where Intelligent Agents are currently being used. Users today want the ability to automatically filter, prioritise and organise their e-mail. Intelligent Agents can facilitate all these functions by allowing mail handling rules to be specified ahead of time, and letting Intelligent Agents operate on behalf of the user according to those rules. It can also be possible to have agents deduce these rules by observing a user's behaviour and trying to find patterns in it. These would be learning agents.

2.8.3 Adaptive User Interfaces

Although the user interface was transformed by the advent of graphical user interfaces, for many, computers remain difficult to learn and use. As capabilities and applications of computers improve, the user interface needs to accommodate the increase in complexity. As user populations grow and diversify, computer interfaces need to learn user habits and preferences and adapt to individuals.

Interface agents can help with both these problems. Intelligent Agent technology allows systems to monitor the user's actions, develop models of user abilities, and automatically help out when problems arise. When combined with speech technology, Intelligent Agents enable computer interfaces to become more human or more "social" when interacting with human users.

2.8.4 Systems and Network Management

Systems and network management is one of the earliest application areas to use Intelligent Agent technology. The movement to client/server computing has intensified the complexity of systems being managed, especially in the area of LANs, and as network centric computing becomes more prevalent, this complexity further escalates. Users in this area (primarily operators and system administrators) need greatly simplified management, in the face of rising complexity.

For example, Intelligent Agents can help filter and take automatic actions at a higher level of abstraction, and can even be used to detect and react to patterns in system behaviour. Furthermore, they can be used to manage large configurations dynamically.

2.8.5 Administrative Management

Administrative management includes both workflow management and areas such as computer/telephony integration, where processes are defined and then automated. In these areas, users need not only make processes more efficient, but also reduce the cost of human agents. Much as in the messaging area, Intelligent Agents can be used to ascertain, then automate user wishes or business processes.

2.8.6 Computer supported collaborative work

Collaboration is a fast-growing area in which users work together on shared documents, using personal video-conferencing, or sharing additional resources through the network. The common denominators are sharing resources and teamwork. Both of these are driven and supported by the move to network centric computing. Not only do users in this area need an

infrastructure that will allow robust, scaleable sharing of data and computing resources, they also need other functions to help them actually build and manage collaborative teams of people, and manage their work products.

2.8.7 Mobile Access

As computing becomes more pervasive and network centric computing shifts the focus from the desktop to the network, users want to be more mobile. Not only do they want to access network resources from any location, they want to access those resources despite bandwidth limitations of mobile technology such as wireless communication, and despite network volatility.

Intelligent Agents which, in this case, reside in the network rather than on the users' personal computers, can address these needs by persistently carrying out user requests despite network disturbances. In addition, agents can process data at its source and ship only compressed answers to the user, rather than overwhelming the network with large amounts of unprocessed data.

2.8.8 Information Management

Information management is an area of great activity, given the rise in popularity of the Internet and the explosion of data available to users. Here, Intelligent Agents are helping users not only with search and filtering, but also with categorisation, prioritisation, selective dissemination, annotation, and collaborative sharing of information and documents.

3. Non-monotonic logic

The intelligent agent paradigm is multi-disciplinary and regroups various computer techniques. Particularly, Artificial Intelligence techniques can be used to endow the agents with human-like reasoning. In this chapter, we draw up a quick overview of a particular field of AI : the non-monotonic logic. In particular, we present briefly Default Logic and Belief Revision, the two kinds of non-monotonic logic we use in the reasoning methods of our Intelligent Agent.

3.1 Introduction

At the basis of non-monotonic logic lies the idea that most of our reasoning is logically wrong.

$$x \text{ is a bird} \Rightarrow x \text{ can fly}$$

This predicate is true in normal conditions, but not always. Situations can be found where the antecedent is true but not the conclusion (for example if X is an ostrich). In these 'special' situations, there is a problem of inconsistency:

$$x \text{ is a bird} \Rightarrow x \text{ can fly}$$
$$x \text{ is an ostrich} \Rightarrow x \text{ cannot fly}$$

So, knowing that an ostrich is a bird, what can we conclude about x if x is an ostrich? Can x fly or not?

Fortunately, the human intelligence has the faculty to elaborate a judicious reasoning when facing incomplete information. This reasoning is often only plausible and can be revised when new information is taken into account. We often include into our belief set statements that have no justification in our initial assumptions beyond the fact that we have no evidence in our belief set to contradict them. This cannot be formalised by classical logic, which is limited to correct and unrevisable reasoning.

Most of our reasoning is not infallible. For example, "knowing that most birds can fly and that Tweety is a bird, I conclude that Tweety can fly". This inference is not correct because it doesn't take into account possible exceptions. Hence, it is uncertain and can be revised. For instance, if we know that "Tweety is an ostrich and ostriches are birds that don't fly", the assertion "Tweety can fly" has to be retracted. Even though this example may seem simple, it is hard to formalise by logic.

Numerous domains of Artificial Intelligence may take advantage of a formal theory of revisable reasoning. The formalisation of some aspects of our elementary faculties of perception (like the vision, which operates by successive approximations and corrections), specialised high-level tasks like diagnosis and dialog analysis (because the communication is often based on implicit information) are some examples.

3.2. Unsuitability of classical logic for formalising revisable reasoning

The deductive systems of classical logic are used to formalise *valid* reasoning. They are not adapted to formalising uncertain and revisable reasoning.

A formal deductive system of classical logic permits to infer conclusions from premises and, hence, defines a relation of interoperability between formulas. This relation is called the "classical inference relation" and is noted $\vdash$. It has the following properties:

- reflexivity
  $\{p_1, \ldots, p_n, q\} \vdash q$
- transitivity
  if $\{p_1, \ldots, p_n\} \vdash r$ and $\{p_1, \ldots, p_n, r\} \vdash q$ then $\{p_1, \ldots, p_n\} \vdash q$
- monotony
  if $\{p_1, \ldots, p_n\} \vdash q$ then $\{p_1, \ldots, p_n, r\} \vdash q$

with $p_1, \ldots, p_n$, q and r designing formulas of the considered logical language.

Clearly, non-monotonic logic does not possess the property of monotony. By definition, the formalisation of revisable reasoning cannot be monotonic. Revising a reasoning means changing the conclusions of that reasoning in presence of new information. Let's say that A is a set of premises from which the conclusion p is inferred. We can revise that reasoning by retracting p when a new information q is added to the premises. This means that:

$$A \vdash p$$
$$A \cup \{q\} \ not \vdash p$$

which is clearly contrary to the definition of monotony.

For building a non-monotonic logic, another inference relation is needed : a non-monotonic inference relation, which permits to draw conclusions that are not *always* true.

The relationship between the classical inference relation $\vdash$ and the classical inference operation *Cn* is :

$$A \vdash x \text{ iff } x \in Cn(A)$$

In other words, $Cn(A)$ equals $\{x : A \vdash x\}$, the set of all classical consequences of A.
The property of monotony of classical logic can be expressed as follows :

$$If\ B \vdash x \text{ and } B \subseteq A \text{ then } A \vdash x$$

The non-monotonic inference relation is denoted by $\vdash\!\sim$ and the non-monotonic inference operation by C. The relationship between the relation and the operation is the same as with the classical inference.

$$A \vdash\!\sim x \text{ iff } x \in C(A)$$

In other words, $C(A)$ equals $\{x : A \vdash\!\sim x\}$, the set of all non-monotonic consequences of A.

3.3. Characteristics of non-monotonic logics

Different logics have been developed in order to formalise revisable reasoning. They are called *non-monotonic* because conclusions that can be drawn from them can decrease when the number of their premises increases (see the monotony property above).

From a same set of initial information, different incompatible sets of possible conclusions can be obtained. This characteristic is called *pluri-extensionality* and these sets are the *extensions*. From a semantic point of view, the conclusions of a revisable reasoning are often simply *consistent*. Hence, *plausible* conclusions, which are not necessarily *certain* or *correct*, can be inferred. And there can exist different *plausible* conclusions that can be drawn from the same initial information depending on the 'guesses' made about the incomplete information.

Formalising revisable reasoning requires a formal framework that is *flexible* enough for working in presence of incomplete information. Moreover, it needs the sufficient *robustness* to enable the modelled reasoning to adjust harmoniously to the evolution of the represented knowledge (evolution of the domain of the discourse or evolution of the conceptualisation of this domain).

In this dissertation, we will focus on the two types of non-monotonic logic we used in developing our agent : Default Logic and Belief Revision.

3.4. Default Logic[1]

In presence of incomplete information, we sometimes draw conclusions that are only *plausible*. We often use laws that are correct in most of the cases, but which admit certain exceptions. Moreover, we allow ourselves to interpret them as absolutely general. For example, if Tweety is a bird, we conclude that Tweety flies, even though we know that there exist some birds that do not fly. The only case where we wouldn't conclude that Tweety flies is when there is a belief in our set of beliefs that forbids us to draw that conclusion, i.e. when that conclusion is inconsistent with our set of beliefs. That kind of reasoning is a *default* reasoning. There is a *default* conclusion that can be drawn if nothing in our set of beliefs forbids it.

Default Logic formalises such default reasoning via special inference rules. Hence, Default Logic distinguishes between two kinds of knowledge: usual predicate logic formulas (the *axioms* or the *facts*) and their inference rules, called *defaults*.

A default rule has the form :

$$\frac{\alpha : \beta}{\gamma}$$

which is read as ' If α and if it is consistent to assume β, then conclude γ'.

[1] All definitions and theorems of this section are taken from [MAREK & TRUSCZYNSKI 1993]

For example, coming back to our example of Tweety, we can express the rule that *generally speaking* birds fly with the default:

$$\frac{Bird(x) : Flies(x)}{Flies(x)}$$

which is read as 'If x is a bird and if it is consistent to assume that x flies, then conclude that x flies'.

The main advantage of a default rule is that it permits to treat all the exceptions without the need to identify all of them. Trying to formalise the exceptions with classical logic looks like this:

$$Bird\ (x) \wedge \neg BrokenWing\ (x) \wedge \neg Ostrich\ (x) \wedge \neg Penguin\ (x) \Rightarrow Flies\ (x)$$

But that rule is still insufficient because there are still more conceivable reasons for a bird not to be able to fly. So we would have to list *all* the possible reasons in the rule. And what's more, we would have to establish that all the preconditions of the rule are true before we could apply it.

Now that we have introduced default logic in an intuitive way, we formalise it by presenting the syntax and the concepts of default proof system and extension.

3.4.1 Syntax and vocabulary

A default δ has the form:

$$\frac{\alpha(x) : \beta 1(x),..., \beta n(x)}{\gamma(x)}$$

where $\alpha(x)$, $\beta_1(x)$, ..., $\beta_n(x)$, $\gamma(x)$ are predicate logic formulas, and where $n > 0$.
α is the *prerequisite*, $\beta_1(x)$, ..., $\beta_n(x)$ the *justifications* and $\gamma(x)$ the *consequent* of the default δ.
$\alpha(x)$ can be denoted by *pre* (δ), $\{\beta_1(x), ..., \beta_n(x)\}$ by *just* (δ) and $\gamma(x)$ by *cons* ($\delta(x)$)

The syntax used for a *prerequisite-free* default : $\alpha(x)$ = true (or *pre* ($\delta(x)$) = $\varnothing$) is noted as follows:

$$\frac{: \beta 1(x),..., \beta n(x)}{\gamma(x)}$$

A *justification-free* default : *just* ($\delta(x)$) = $\varnothing$

$$\frac{\alpha(x) :}{\gamma(x)}$$

A default δ is said to be *closed* iff $\alpha(x)$, $\beta_1(x)$, ..., $\beta_n(x)$ and $\gamma(x)$ do not contain any *free* variable. In that case, $\alpha(x)$, $\beta_1(x)$, ..., $\beta_n(x)$ and $\gamma(x)$ are simply denoted by α, β_1, ..., β_n and γ.

The *free* variables of a default are interpreted as universally quantified (their reach extends to all the elements of the default). A default that is not *closed* is *open*. An *open* default represents a general inference schema. An instance of an *open* default is a *closed* default obtained by replacing all the *free* variables of the *open* default by terms.

A *default theory* is a pair (W, D) constituted of a set W of predicate logic formulas (the *axioms* or the *facts*) and a set D of *defaults*. A default theory is said to be closed iff all the defaults of D are closed.

3.4.2 Default proof system

Definition 1 *Let D be a set of inference rules. By a rule proof system (or rule system) PC+D, we mean the proof system obtained by adding the rules in D to the proof system of propositional logic. A derivation of a formula φ from a set of formulas W in the system PC + D is a finite sequence of formulas $\varphi_1, \ldots, \varphi_n$ such that:*

1. $\varphi_n = \varphi$.
2. *For every $i \leq n$ at least one of the following conditions holds :*
   - *(a) φ_i is a substitution instance of an axiom of propositional logic.*
   - *(b) $\varphi_i \in W$.*
   - *(c) For some j, $k < i$, φ_i follows from φ_j and φ_k by modus ponens.*
   - *(d) For some $j_1, \ldots, j_k < i$, the rule $\frac{\varphi j_1, \ldots, \varphi j_k}{\varphi_i}$ belongs to D.*

By $C_n^D(W)$ we denote the set of all the formulas that possess a derivation from W in $PC+D$. Some simple but useful properties of the operator C_n^D are given in the following theorem.

Theorem 1 *Let D be a set of inference rules. For every set $W \subseteq L$:*

1. $C_n(W) \subseteq C_n^D(W)$.
2. $C_n^D(C_n^D(W)) = C_n^D(W)$.

In other words, $C_n^D(W)$ is closed under propositional provability and all the rules in D.

We will provide a characterisation of the operator C_n^D in the case when D and W are finite. It provides a finite representation for $C_n^D(W)$ and is useful in determining whether a formula φ belongs to $C_n^D(W)$ or not. To this end, we need to define B^D called the *base operator*.

Definition 2 *For a set D of defaults and a theory W put*

$$B^D(W) = W \cup \{\gamma : \frac{\alpha}{\gamma} \in D \text{ and } \alpha \in C_n(W)\}.$$

Theorem 2 *For every theory $W \subseteq L$ and set of rules D :*

$$C_n^D(W) = C_n(B^D{}_{\varpi}(W)).$$

Definition 3 *Let D be a set of defaults. By a default proof system (or default system) PC+D we mean the proof system of propositional logic extended by the set of defaults D. The concept of a proof in a default proof system PC + D is defined relative to a theory* $S \subseteq \mathsf{L}$. *An S-derivation (S-proof) of a formula* φ *from W in PC + D is a finite sequence* $\varphi_1, \ldots, \varphi_n$ *such that* $\varphi_n = \varphi$ *and, for every i,* $1 \leq i \leq n$, *at least one of the following conditions holds :*

1. $\varphi_i \in W$.
2. φ_i *is a substitution instance of an axiom of propositional calculus.*
3. φ_i *is the result of applying modus ponens to formulas* φ_j, φ_k *for some* $j, k < i$.
4. *There is a default* $d = \dfrac{\alpha(x) : \beta_1(x), \ldots, \beta_k(x)}{\varphi_i(x)}$

   such that $d \in D$, $\alpha(x) = \varphi_j(x)$ *for some* $j < i$, *and* $\neg \beta_1(x) \notin S, \ldots, \neg \beta_k(x) \notin S$.

The theory S is called a *context*. By $C_n^{D,S}(W)$ we denote the set of all the formulas having an *S*-proof from *W* in *PC+D*.

The notion of a default proof system is a substantial departure from the standard notion of a proof system. Default systems are endowed with a whole family of consequence operators parameterised with a context *S*, while standard proof systems, like rule proof systems, have only one consequence operator associated with them. This similitude of consequence operators is a formal reflection of the fact that default rules allow an agent to make assumptions when full information is not available. Consequently, depending on the assumptions the agent makes, different belief (consequence) sets can be produced for a theory *W* of initial assumptions.

Inference rules can be treated as justification-free defaults and, consequently, a default proof system can be regarded as a natural generalisation of a rule proof system as will be shown now.

First, for a set *D* of defaults, we define

$$D_m = \{\frac{\alpha}{\gamma} : \frac{\alpha :}{\gamma} \in D\}.$$

Theorem 3 Let *D* be a set of justification-free defaults. For every context $S \subseteq \mathsf{L}$ and for every theory $W \subseteq \mathsf{L}$

$$C_n^{Dm}(W) = C_n^{D,S}(W).$$

In other words, all consequence operators associated with the default system *PC + D* coincide and are equal to the consequence operator of the rule system $PC + D_m$.

3.4.3 Extensions

The operational semantics of Default Logic are defined in terms of so-called *extensions*, sets of beliefs one may hold about the domain described by the default theory under consideration. An extension is an ensemble of information that includes all what can be inferred, by classical logic rules or by defaults. Extensions are obtained by applying defaults as long as possible

without running into inconsistencies. It means that if we find out that a default should not have been applied, we backtrack and try some alternative.

For a given default theory $T = (W, D)$, let $\varphi = (\delta_0, \delta_1, \ldots)$ be a sequence of defaults from D. There should not be any multiple occurrence in this sequence because that would mean that we apply a default more than once. This is useless because no additional information would be gained. One sequence leads to one extension. As there can be different sequences to apply the defaults, there can be more than one extension for a given theory.

Definition 4 *Let (D, W) be a default theory. We say that a theory S is a default extension for (D, W) if*

$$S = C_n^{D,S}(W).$$

Extensions can be viewed in two ways. First, they are contexts that determine which consequence operators in the family $\{ C_n^{D,S} : S \subseteq \mathsf{L} \}$ can be used by an agent to construct belief sets. At the same time, extensions are exactly the belief sets the agent computes in this manner.

It follows from definition 4 that the process of finding an extension consists of two phases. In the first phase we guess the context S. In the second, we check if S satisfies the equation $S = C_n^{D,S}(W)$. The whole process simplifies significantly if all defaults in a default theory are justification-free. Each such default theory possesses exactly one extension and it can be found in a constructive manner, without the need to guess a context first.

Theorem 4 *Let D be a set of justification-free defaults. For every theory $W \subseteq \mathsf{L}$, the theory $C_n^{Dm}(W)$ is the only extension for (D, W).*

Definition 5 *Given a set of default rules D and a subset $S \subseteq \mathsf{L}$, the reduct of D with respect to S, denoted by D_s, is the set of inference rules defined as follows :*

$$D_s = \{ \frac{\alpha}{\gamma} : \frac{\alpha : M\beta_1, \ldots, M\beta_n}{\gamma} \in D \text{ and } \neg\beta_1 \notin S, \ldots, \neg\beta_n \notin S \}.$$

The reduct D_s contains exactly the information needed for constructing S-derivations in the default system $PC + D$. It allows us to characterise the operator $C_n^{D,S}$ of a default proof system in terms of the consequence operator of the rule proof system.

Theorem 5 *For every context S, for every theory W and for every set of default D,*

$$C_n^{D,S}(W) = C_n^{Ds}(W).$$

As corollaries to this theorem, we obtain several useful properties and characterisations of extensions.

Corollary 1 *A theory S is an extension for a default theory (D, W) if and only if*

$$S = C_n^{Ds}(W).$$

Corollary 2 *A theory S is an extension for a default theory (D, W) if and only if*

$$S = C_n(B^{Ds}{}_{\varpi}(W)).$$

The following examples shows how corollary 2 can be used to generate the extensions of a given default theory.

Example

Let $D = \{\frac{p : \neg q}{r}, \frac{p : \neg r}{q}\}$ and $W = \{p\}$.

We have four candidates for extensions :

$S_1 = C_n(\{p\})$
$S_2 = C_n(\{p,q\})$
$S_3 = C_n(\{p, r\})$
$S_4 = C_n(\{p, q, r\})$

First, we compute :

$D_{S1} = \{\frac{p}{r}, \frac{p}{q}\}$

$D_{S2} = \{\frac{p}{q}\}$

$D_{S3} = \{\frac{p}{r}\}$

$D_{S4} = \varnothing$

Next, we compute :

$B^{Ds1}{}_{\varpi}(W)) = \{p, q, r\} \neq S_1$
$B^{Ds2}{}_{\varpi}(W)) = \{p, q\} = S_2$
$B^{Ds3}{}_{\varpi}(W)) = \{p, r\} = S_3$
$B^{Ds4}{}_{\varpi}(W)) = \{p\} \neq S_4$

Hence, S_2 and S_3 are extensions for (D, W), while S_1 and S_4 are not.

3.5. Belief Revision[1]

The purpose of Belief Revision is to lay the foundations of principled mechanisms for modifying a knowledge base in a rational and coherent way. It is useful for an intelligent system that has to change its knowledge base when it acquires new information. This is particularly important when that new information is in conflict with the current knowledge base.

[1] All definitions and theorems of this section are taken from [ANTONIOU 1997]

Change functions are based on the "Principle of Minimal Change" that states that as much information should be conserved as is possible in accordance with an underlying preference relation. The simplest change function is *expansion*. It is used when there is no conflict at all and the new information can simply be incorporated in the knowledge base. In order to incorporate new information that is inconsistent with the knowledge base, the system must *decide* what information it is prepared to give up or to revise. The change functions used to deal with these contradictions are more complex. These are *contraction* and *revision*.

Change functions can be described *axiomatically* using rationality postulates, or *constructively* using preference relations. The rationality postulates are properties that one would expect rational change functions to satisfy. They embody the "Principle of Minimal Change", and act as integrity constraints for change functions. They do not uniquely determine a change function, rather their purpose is to identify the set of possible new knowledge bases that might result of the Belief Revision. Those postulates are called the *AGM postulates*, from the name of their developers: Alchourron, Gärdenfors and Makinson.

In order to single out an individual function, it is usual to make use of a preference relation such as an *epistemic entrenchment ordering* that provides the extra-logical information required to make necessary choices concerning what information should be given up.

Example

Let K be a knowledge base composed of the following formulas:

(1) Bird (Tweety)
(2) Bird (x) => Flies (x)
(3) Flies (Tweety)

Then comes a new information:

(4) ¬Flies (Tweety)

In order to incorporate the new formula that is inconsistent with the knowledge base, the system must *decide* what information it is prepared to give up. How K is changed depends on that decision. Either we drop (2) and (3) and then K becomes:

Bird (Tweety)
¬Flies (Tweety)

Or we give up (1) and (3) and then K becomes:

Bird (x) => Flies (x)
¬Flies (Tweety)

Or we abandon (1), (2) and (3). Then K becomes:

¬Flies (Tweety)

3.5.1 The AGM rationality postulates

The new knowledge base, built from the old knowledge base and the new information, always contains this new information. Hence, the only way the knowledge base can become inconsistent is if the new information is inconsistent. Moreover, changes are minimal, e.g. the least amount of information is lost and the least amount of information is gained.

We will now discuss the rationality postulates for the 3 main change functions: expansion, contraction and revision.

Expansion

Expansion is the simplest change to a knowledge base. It involves the acceptance of information without removing any other information in the knowledge base. The expansion of a theory T with respect to a sentence φ is the logical closure of T and φ. Let's say that L is a given language and that K_{L} is the set of all the theories of that language. Then, an expansion function is a function from K_{L} x L to K_{L}, mapping (T, φ) to $T^{+}{}_{\varphi}$ where $T^{+}{}_{\varphi} = C_n(T \cup \{\varphi\})$.

Contraction

Contraction of a knowledge base involves the retraction of information. A sentence is retracted from the knowledge base without adding any new facts. In order for the resulting system to be closed under logical consequences some other sentences must be given up too. The difficulty is in determining which sentences should be given up.

A contraction of T with respect to φ involves the removal of a set of sentences from T so that φ is no longer implied, provided φ is not a tautology. A contraction function is a function from K_{L} x L to K_{L} , mapping (T, φ) to $T^{-}{}_{\varphi}$ which satisfies the following postulates:
For any $\varphi, \gamma \in \mathsf{L}$ and any $T \in K_{\mathsf{L}}$:

(1) $T^{-}{}_{\varphi} \in K_{\mathsf{L}}$
(2) $T^{-}{}_{\varphi} \subseteq T$
(3) If $\varphi \notin T$ *then* $T \subseteq T^{-}{}_{\varphi}$
(4) If not $\vdash \varphi$ *then* $\varphi \notin T^{-}{}_{\varphi}$
(5) $T \subseteq (T^{-}{}_{\varphi})^{+}{}_{\varphi}$ *(recovery)*
(6) If $\vdash \varphi \leftrightarrow \gamma$ *then* $T^{-}{}_{\varphi} = T^{-}{}_{\gamma}$
(7) $T^{-}{}_{\varphi} \cap T^{-}{}_{\gamma} \subseteq T^{-}{}_{\varphi \wedge \gamma}$
(8) *If* $\varphi \notin T^{-}{}_{\varphi \wedge \gamma}$ *then* $T^{-}{}_{\varphi \wedge \gamma} \subseteq T^{-}{}_{\varphi}$

Revision

Revision attempts to change a knowledge base as little as possible in order to incorporate new information. A new sentence that is inconsistent with the knowledge base is added, but in order to maintain consistency in the resulting knowledge base, some of the old sentences are deleted.

A revision function is a function from K_{L} x L to K_{L} , mapping (T, φ) to $T^{*}{}_{\varphi}$ which satisfies the following postulates:

For any $\varphi, \gamma \in \mathsf{L}$ and any $T \in K_L$:

(1) $T^*_\varphi \in K_L$
(2) $\varphi \in T^*_\varphi$
(3) $T^*_\varphi \subseteq T^+_\varphi$
(4) If $\neg\varphi \notin T$ *then* $T^+_\varphi \subseteq T^*_\varphi$
(5) If $T^*_\varphi = \bot$ *then* $\vdash \neg\varphi$
(6) If $\vdash \varphi \Leftrightarrow \gamma$ *then* $T^*_\varphi = T^*_\gamma$
(7) $T^*_{\varphi \wedge \gamma} \subseteq (T^*_\varphi)^+_\gamma$
(8) If $\neg\varphi \notin T^*_\varphi$ *then* $(T^*_\varphi)^+_\gamma \subseteq T^*_{\varphi \wedge \gamma}$

3.5.2 Relationships between change functions

There exist relationships between the different change functions. Here are two theorems that illustrate them. Theorem 6 states that a revision function can be expressed in terms of contraction and expansion functions whereas theorem 7 says that a contraction can be expressed in terms of a revision function.

Theorem 6 *If* $^-$ *is a contraction function and* $^+$ *the expansion function, then* * *defined by the Levi Identity below defines a revision function.*

$$T^*_\varphi = (T^-_{\neg\varphi})^+_\varphi$$

Theorem 7 *If* * *is a revision function, then* $^-$ *defined by the Harper Identity below defines a contraction function.*

$$T^-_\varphi = T \cap T^*_{\neg\varphi}$$

3.5.3 Epistemic entrenchment orderings

The rationality postulates for contraction and revision do not provide a mechanism for defining a particular function, they only describe classes of functions. For any theory, there can be a lot of functions that satisfy the postulate. So in order to single out a unique one, an additional structure is necessary : a preference relation such as an *epistemic entrenchment ordering*, which is a total pre-order on formulas.

All the sentences in our knowledge base are not of equal value. Some have a higher degree of *epistemic entrenchment* than others do. This means that they are more important than others when planning future actions. The degree of entrenchment will be used to choose what belief to discard when a contraction or a revision is carried out. It represents the plausibility of a belief, or the degree of attachment to a belief. When a belief set is revised or contracted, the sentences that are given up are those having the lowest degree of epistemic entrenchment.

Example

Let K be a knowledge base composed of the following formulas:

(1) Bird (Tweety) of degree 0.9
(2) Flies (Tweety) of degree 0.8
(3) Bird (x) => Flies (x) of degree 0.4

Than comes a new information:

(4) ¬Flies (Tweety) of degree 0.95

As (3) has the lowest rank, it will be given up and K will become :

Bird (Tweety)
¬Flies (Tweety)

The notation "$\varphi \leq \gamma$" will be used as a shorthand for "*γ is at least as epistemically entrenched as φ*". The notation "$\varphi < \gamma$", used as a shorthand for "*γ is epistemically more entrenched than φ*", is defined as "$(\varphi \leq \gamma) \wedge \neg(\gamma \leq \varphi)$".

Definition 6 *Given a theory T of* $\mathcal{L}$, *an epistemic entrenchment related to T is any binary relation* $\leq$ *on* $\mathcal{L}$ *satisfying the conditions below :*

(1) If $\varphi \leq \gamma$ *and* $\gamma \leq \delta$, *then* $\varphi \leq \delta$ *(transitivity)*
(2) If $\varphi \vdash \gamma$, *then* $\varphi \leq \gamma$ *(dominance)*
(3) For any φ *and* γ, $\varphi \leq \varphi \wedge \gamma$ *or* $\gamma \leq \varphi \wedge \gamma$ *(conjunctiveness)*
(4) When $T \neq \bot$, $\varphi \notin T$ *iff* $\varphi \leq \gamma$, *for all* $\gamma \in \mathcal{L}$ *(minimality)*
(5) If $\varphi \leq \gamma$ *for all* φ, *then* $\vdash \gamma$ *(maximality)*

We just give here the justification for one postulate to show the relationship between these postulates and the change functions : the justification for (2) is that if φ logically entails γ, and either φ or γ must be retracted from T, then it will be a smaller change to give up φ and to retain γ rather than to give up γ, because then φ must be retracted too.

The two following theorems provide us with a constructive method for building contraction and revision functions from an epistemic entrenchment ordering.

Theorem 8 Let T be a theory of $\mathcal{L}$. For every contraction function $^-$ for T there exists an epistemic entrenchment $\leq$ related to T such that (E^-) below is true for every $\varphi \in \mathcal{L}$. Conversely, for every epistemic entrenchment $\leq$ related to T, there exists a contraction function $^-$ such that (E^-) is true for every $\varphi \in \mathcal{L}$.

$$(E^-) \quad T^-_\varphi = \begin{cases} \{\gamma \in T : \varphi \leq \varphi \vee \gamma\} & \text{if not } \vdash \varphi \\ T & \text{otherwise} \end{cases}$$

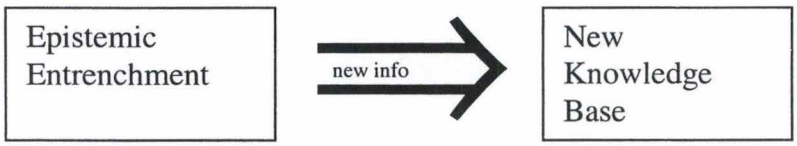
Theorem 9 Let T be a theory of L. For every revision function * for T there exists an epistemic entrenchment $\preceq$ related to T such that (E^*), below, is true for every $\varphi \in \mathsf{L}$. Conversely, for every epistemic entrenchment $\leq$ related to T, there exists a revision function * such that (E^*) is true for every $\varphi \in \mathsf{L}$.

$$(E^*) \quad T^*_\varphi = \begin{cases} \{\gamma \in T : \neg\varphi \leq \neg\varphi \vee \gamma\} & \text{if not} \vdash \neg\varphi \\ \perp & \text{otherwise} \end{cases}$$

With such theorems, it is easy to construct, from a given theory (belief set) and a given new belief, the theory that forms the belief set after a contraction or a revision.

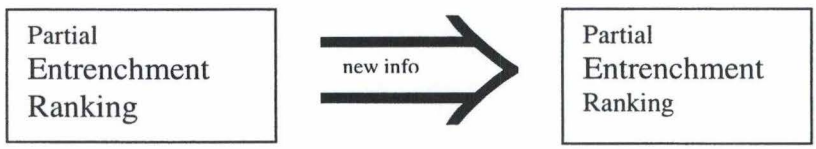
3.5.4 Implementation of Belief Revision

When one attempts to implement the process of Belief Revision, two major obstacles arise. The first difficulty arises because an epistemic entrenchment ordering ranks sentences in a possibly infinite logical theory. It is obvious that ranking an infinite number of sentences presents a serious representation problem for a computer-based implementation. The fact that change operators do not propagate an epistemic entrenchment ordering constitutes the second problem.



The epistemic entrenchment ordering is lost in the process of change, and as a consequence the iteration of a change function is not naturally supported. In many applications information systems are bombarded with new information and this makes modelling the iteration of change essential.

The solution to these two problems is a finite representation of a finitely representable epistemic entrenchment ordering, a *finite partial entrenchment ranking*, coupled with the mechanism of transmutations that revise the ranking and not just the beliefs.



A *finite partial entrenchment ranking* grades the content of a finite knowledge base according to its epistemic importance. It maps a finite set of sentences to rational numbers. The higher the value assigned to a sentence the more strongly believed it is.

Definition 7 *A finite partial entrenchment ranking is a function E from a finite subset of sentences into the interval [0, 1] such that the following conditions are satisfied for all* $\varphi \in dom(E)$:

(1) $\{\gamma \in dom(E) : E(\varphi) \leq E(\gamma)\}$ *not* $\vdash \varphi$ *if* φ *is not a tautology*
(2) If $\vdash \neg\varphi$, *then* $E(\varphi) = 0$
(3) $E(\varphi) = 1$ *iff* $\vdash \varphi$

These conditions are interpreted like this : (1) states that sentences cannot entail lower ranked sentences, (2) says non-beliefs (i.e. inconsistent sentences) are mapped to zero, and (3) states that tautologies are maximal.

Definition 8 *The explicit information content is defined as* $\{\varphi \in dom(E) : E(\varphi) > 0\}$ *and denoted by exp(E). The implicit information content is defined as Cn(exp(E)) and denoted by content(E).*

By definition of *Cn,* which is the classical logic closure, $exp(E) \subseteq content(E)$.

Definition 9 *Let* φ *be a non-tautological sentence. Let E be a finite partial entrenchment ranking. We define the degree of acceptance of* φ *to be :*

$$Degree(E, \varphi) = \begin{cases} \text{largest j such that } \{\varphi \in \exp(E) : E(\varphi) \geq j\} \mid -\varphi \text{ if } \varphi \in content(B) \\ 0 \qquad \text{otherwise} \end{cases}$$

With such a definition, it is easy to design an algorithm to calculate the degree of acceptance of a sentence given the information encoded in a finite partial entrenchment ranking. That algorithm would attempt to prove this sentence using the sentences assigned the largest value in the range of E, say n. If it succeeds, the sentence is assigned the degree n. If not the algorithm has to try to prove it using sentences assigned the next largest degree, say m, as well as using sentences assigned n. If it succeeds the degree is m, etc.

Transmutations provide a constructive basis for an iterated revision. A (φ, i)-*transmutation* is a process that revises a partial entrenchment ranking E to produce a new partial entrenchment ranking, denoted $E^*(\varphi, i)$. It modifies E by assigning i the degree of acceptance of φ such that content($E^*(\varphi, i)$) satisfies the AGM postulates. For $i > 0$, content($E^*(\varphi, i)$) is a revision, and for $i = 0$, it is a contraction.

3.5.5 Types of Transmutation

There exist several types of transmutations. In this dissertation, we just present briefly two of them : standard adjustment and maxi-adjustment. In the belief revision module of our agent, we use Maxi-Adjustment because it is more appropriate to the way we want it to behave.

Standard Adjustment

This type of transmutation is based on the standard entrenchment construction :
if φ is the new information, γ is retained iff degree (φ) < degree(γ). It imposes an absolute measure of minimal change.

Maxi-adjustment

Spohnian reasons

Definition 10 *γ is a reason for φ iff raising the epistemic rank of γ would raise the epistemic rank of φ.*

If γ, φ in exp(E) and if we preserve the proprieties of a partial entrenchment ranking, then γ is a reason for φ iff $\gamma \Rightarrow \varphi > \varphi$.

Maxi-adjustments specify reasons and during the contraction of φ, we retain γ if we cannot derive that γ is a reason for φ. It adopts a policy of maximal information inertia, i.e. information stays at its current rank unless there is a reason to change it.

Comparison

For standard adjustment, one calculates what has to be retained. What is retained is 'nothing but what has been found'. For maxi-adjustment, one calculates what has to be given up and one retains 'everything but what has been found'. Hence, the explicit beliefs obtained via maxi-adjustment always contain the explicit beliefs of an adjustment.
For example, let's take the following ranking :

$a \vee b$ of degree 0.9
$a \vee d$ of degree 0.5
$-c \vee -b$, c, d, e of degree 0.3

If we make a revision with $-a$, standard adjustment would remove c, $-c \vee -b$, and e, because they cannot be proven from $-a$, $a \vee b$, $a \vee d$. On the contrary, d can and would thus be kept. Maxi-adjustment, however, would keep e, and removes only $-c \vee -b$ and c. It would find the first contradiction at the third rank, and determines that the only subset of the third rank that is responsible is : $\{-c \vee -b, c\}$, and so these would be the only elements removed.

4. Conclusion

The first part of this dissertation presents the theoretical basis on which we build our application. We first examine the current state of the intelligent agent discipline. Then we expose the theoretical aspects of the non-monotonic logic that our agent made use of to reason.

Concerning intelligent agent, we have seen that issues such as agent's architecture, communication or agent-oriented languages have been already well addressed. Indeed, candidate standards that treat these issues already appear. Standards are of great help, but it will take quite some time before these are totally efficient, accepted and used. When that happens, standards will help speed up developments but the lack of a perfect standardisation is likely to slow down developments in the meantime. In the future, there will be perhaps an *agent protocol* on the Internet just as there are protocols for the hypertext transfer or the file transfer.

Other important issues, such as security, user privacy, and many ethical and juridical issues, have not yet been addressed and tackled, or only partially. Expectations are that, within several years, enough of these issues will have been sufficiently dealt with. Hence, much has still to be done to make the Intelligent Agent technology perfect, or at least more mature...

During the last couple of years, agents have been the critics' "moving target". They have been incorporated into future doom scenarios, where they are used to spy on Internet users, and where they turn people into solitary creatures, that live their life inside their own little virtual reality. Agents were said to be the latest hype, and - as a technique - had not much to offer. But in the latest years, these people had to change their opinion with the appearing of really useful applications based on the Intelligent Agents technique. Although this technique is still young and has received many critics, it looks currently promising. The success is mainly due to an interesting and powerful aspects of Intelligent Agents : their ability to communicate with other agents, other applications and - of course - with humans.

Just as the Intelligent Agent technology, the Artificial Intelligence discipline has gone trough a period of critics and scepticism. Now, techniques such as non-monotonic logic seem to get mature. Default logic and belief revision permit to formalise revisable reasoning which cannot be treated by classical logic. It is thus a real breakthrough in the search of imitating the human reasoning. Though, the technique is still far from being popular compared to the Intelligent Agent. Maybe the Intelligent Agents have the to address to a larger variety of domains of application. With the growth of the amount of information available on the Internet, Intelligent Agents will become more and more indispensable whereas Artificial Intelligence and non–monotonic logic in particular are more specific to specialised applications that won't meet directly the same success.

The time seems to have come to confront these two disciplines to build systems that are as useful as Intelligent Agents are and that behave as 'intelligently' as AI-based reasoning permits. Maybe such systems will be the Intelligent Agents of tomorrow...

PART II

INTELLIGENT AGENTS AND NON-MONOTONIC LOGIC : AN APPLICATION

1.Introduction

In the second part of our dissertation, we proceed with the more concrete side of our researches. Our initial intention was to build an Intelligent Agent that could use non-monotonic reasoning in order to behave more "intelligently" in some situations. We decided to use non-monotonic logic, more specifically Belief Revision and Default Logic, as the reasoning methods of our agent. But it was not easy to find a domain of application where we could take full advantage of these reasoning techniques. Indeed, there are not a lot of applications involving non-monotonic logic as reasoning method for Intelligent Agents. After many hesitations, we finally opted for a travel agent that should be able to find flight tickets that match the best the preferences expressed by a traveller. Will such agents be used by travel agencies or directly by the travellers? Will the emergence of these agents mean the disappearance of travel agencies? These are some of the questions we try to answer in the chapter 2 that introduces the travel industry and the position Intelligent Agents could hold in it.

In order to perform its search for the "best ticket", our travel agent should be able to gather the information needed through directly accessing the airlines' databases or through communicating with other Intelligent Agents representing these airlines. We chose the second solution because it seemed to be more interesting and more powerful. Hence, we had to build a complete multi-agent system that consists of agents representing the travellers and agents representing the airlines. The chapter 3 presents the characteristics of this system.

In our application, the inter-agent communication and other functionalities are simplified through the fact that our agents were developed in the same programming language, namely Jack, an agent-oriented language based on Java. We discuss it briefly in chapter 4.

Non-monotony is used in two ways by our travel agent. First, Default Logic permits to deal with incomplete information coming from the Intelligent Agents representing the airlines. Second, Belief Revision is used to treat changes in the tickets offered by the airlines or in the preferences of the user. The advantages of using Default Logic in our agent' s reasoning methods are clear, but it is not that obvious for Belief Revision. Moreover, we did not use Belief Revision in a classical way, i.e. we did not maintain a consistent belief set by revising it as soon as new beliefs appear. We used Belief Revision to implement a kind of selection system. Its goal is to single out a ticket on the basis of its characteristics and of preferences expressed by the user.

In chapter 5, we introduce three tools we use to deal with non-monotonic reasoning. These tools are Vader (a theorem prover), Hades (that treats Default Logic) and Saten (that deals with Belief Revision).

Afterwards, in chapter 6, we describe the external architecture of our system as well as the internal architecture of the agents that compose it. These agents are of two types : the TravelAgents which are agents acting on behalf of a traveller and CompanyAgents which represent the airlines.

To close this second part of our dissertation, we focus in chapter 7 on the real implementation. First, we introduce how the graphical interface of our application looks like and explain how to use it. Next, we discuss the way our TavelAgent uses Default Logic and Belief Revision. Finally, we show how to build the different objects that form a Jack agent by analysing the code of some of them.

To summarise, three steps have to be realised to build our agent system. The first step is analysing the characteristics of the travel agent (chapters 2 and 3). The second step is designing a framework (chapter 6) respecting the constraints of the tools (chapters 4 and 5). Finally, the last step is implementing the application (chapter 7).

2. Travel Agents and the travel industry

In this chapter, we present the domain of application we chose for our agent. We present the travel industry and the role software travel agents can play in it. We will first introduce the travel industry in general. Afterwards, we explain the properties that a travel agent should have. We also present the different actors present in this industry and the possible scenarios involving them. We conclude with a rapid view of what the future of the travel agents could look like.

2.1 Industry background[1]

In many ways, the travel industry is the best example of an industry profoundly transformed by technology. Historically, this industry has been an early adopter of new technologies, for instance Computer Reservation Systems (CRS). As technology becomes more pervasive, traditional consumers begin to use tools formerly reserved for travel professionals. In the case of CRS, consumers who have access to similar systems through their home computers and open networks (primarily the Internet) can now take over some functions traditionally performed by travel agents. This should also be combined with the current increased demand for travel (foreseen to continue). Therefore, it seems to be an extraordinary period of time, where drastic changes are inevitable.

Faced with this need for change, the travel industry currently relies on an outdated distribution network, essentially relying on third parties. In recent years, travel agencies made use of a specialised technology infrastructure and of specific knowledge, to justify their cost. New technologies are progressively rendering this infrastructure obsolete as providers begin to understand how to deliver information directly to their customers, through phone, fax, electronic mail and increasingly, through multimedia interactive systems. This endangers the travel agencies, who will need to reposition themselves.

The landscape of the industry as we know it today will profoundly be affected. Some players will disappear, new players will emerge, and all of the current actors will have to change in order to survive. In which direction, whom, how, and how much are still open questions. Every organisation currently active in the industry will be affected, from airlines to travel agencies, from large corporations to small and medium enterprises, and certainly individual travellers.

2.2 Basic facilities and properties of a travel agent

Obviously, every traveller has at least one point of departure and one point of arrival (and maybe a date required for travel) which he/she specifies to the agent. Other attributes can be necessary to make the selection of flights more suitable to the customer's desires. This would include : economy class or business class, one-way or return booking, direct flight or flight with connecting city, some services that the customer desire in the airplane (e.g. champagne, television, smoker seat), ...

[1] based upon [BLOCH & SEGEV 1996]

Once the agent knows what the customer requests, the next obvious step consists in actually booking all the travel details required. Availability is the main concern for both the customer and the agents. This means that every reservation system should allow the user to check for availability of travel seats.

More complete travel agents should offer other services such as hotel reservation or car hire as well. A travel agent's life can be made very complicated if he/she has to use separate unrelated systems to book flight tickets, make hotel arrangements and arrange for car-rental agreements. Fortunately, most systems, including Sabre[1] integrate these services, thus making everything more efficient and reliable.

Up to now, we discussed the main features required to make a reservation system work properly and satisfactorily to meet most customer's demands. However, as more and more up-to-date information can be obtained on-line, these systems can integrate more facilities allowing one to access this material, such as currency, visa and health information, weather, tourist attractions,... The next step would be a complete personal Intelligent Agent that would manage the traveller's trips, as well as his/her mail, a personally adapted search engine, his/her call schedule, ... This agent could even adapt to its user by learning his/her way of filtering his/her mail, his/her preferences when he/she searches something on the Web, ...

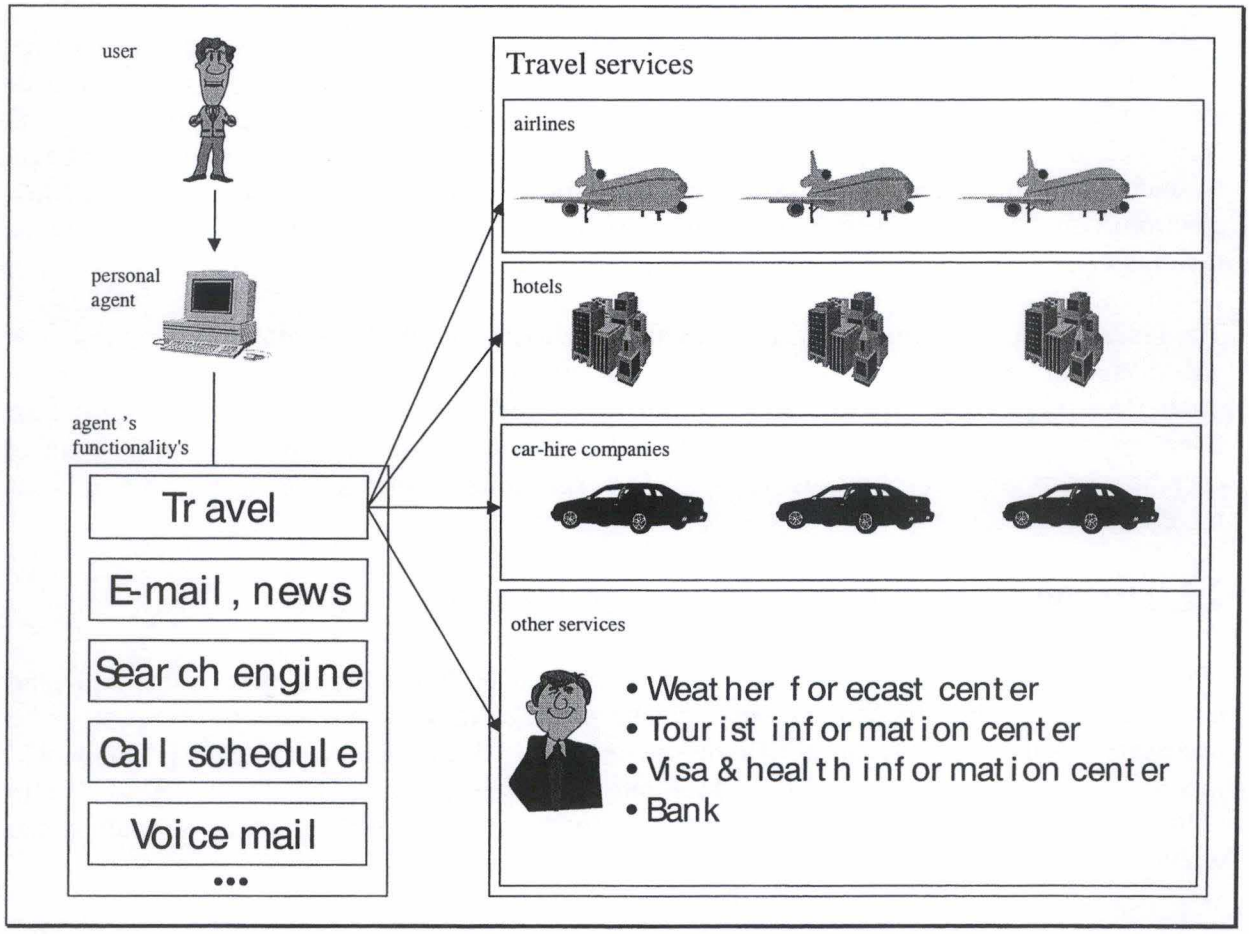


Figure 7 : What a personal agent could look like

[1] http://www.sabre.com

2.3 The actors[1]

The travel industry can be analysed using an industry value chain, as shown below in Figure 8. In its simplest form, the industry consists of suppliers (or vendors), distributors, and customers.

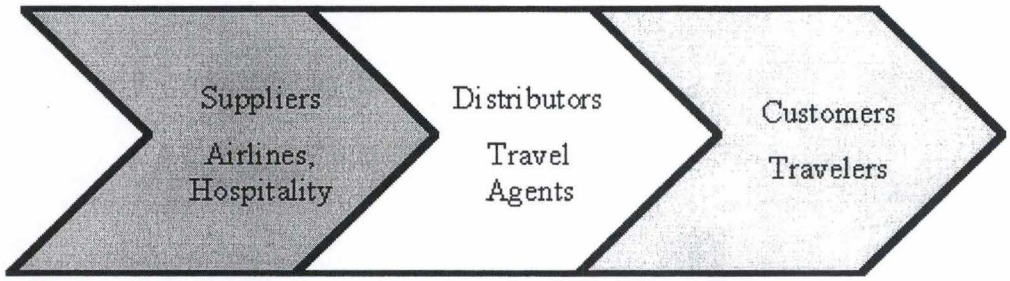


Figure 8 : The travel industry value chain[2]

2.3.1 Suppliers / Vendors

Suppliers of travel-related services include transportation and hospitality service providers. These include airlines, railways, car-rental agencies, cruise lines, hotels, and a collection of entertainment service providers.

2.3.2 Distributors

Travel agents represent the primary distribution channel in the travel industry. They are information brokers whose primary purpose is to act as a clearinghouse that brings travel service suppliers in contact with their customers. Travel agents make money from commissions on a variety of products (flight, tours, hotel, insurance, ...).

2.3.3 Customers

Customers are travellers, consisting primarily of business and leisure travellers. These two segments have different needs, different levels of price sensitivity, and exert different amounts of influence over suppliers and distributors.

2.4 Possible Scenarios[3]

We will now consider three possible scenarios to see what effect they could have on the structure of the travel industry. The three scenarios are illustrated in Figure 9.

---

[1] based upon [HEILMANN et al. 1995]
[2] from [HEILMANN et al. 1995]
[3] based upon [HEILMANN et al. 1995]

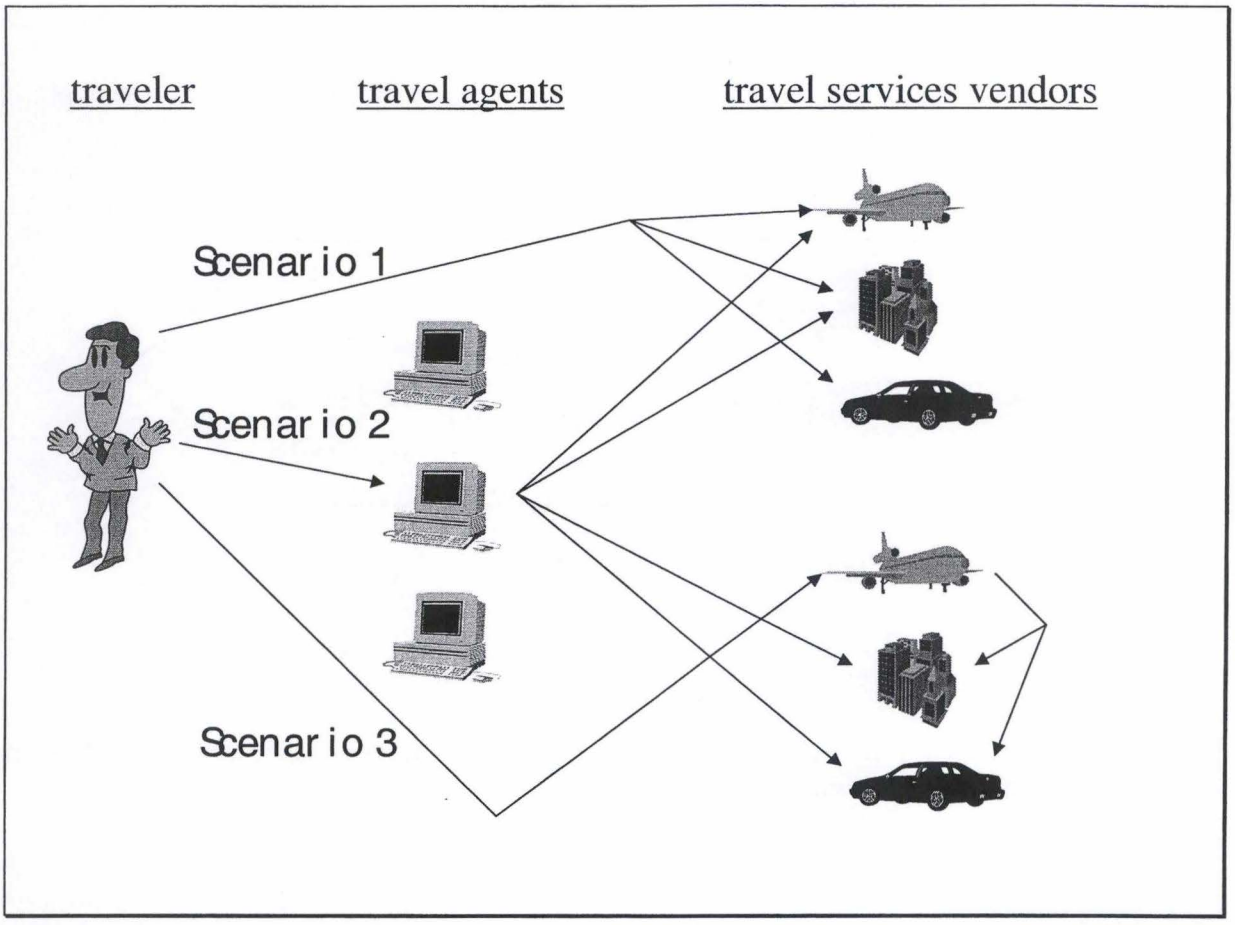


Figure 9 : The three possible scenarios

2.4.1 External Vendor provides travellers their own Intelligent Agents

Our first scenario embodies a system with an agent that belongs to the user and books travel tickets based on his/her personal preferences. The agent would be provided by an external vendor who has convinced travel vendors to co-operate.

The agent is owned and paid for by the traveller. He/she uses such an Intelligent Agent as a service and pays for the search cost as opposed to a percentage of any transaction cost. The agent is seen as being independent and not linked to a particular vendor as in scenario 3.

Travel service vendors have an incentive to take part, since this alternative allows them to dispense with the services of a traditional travel agent altogether. It also allows them to personalise their services.

2.4.2 Travel Agents develop their own Intelligent Agents

Travellers expect travel agents to represent all vendors as opposed to just one vendor. The scenario of travel agency as a clearinghouse of travel related services is still a viable business model. In this scenario, the individual travel agents or groups of travel agents own Intelligent Agents. Cost savings from operations and perhaps a percentage of transaction costs will pay for the Intelligent Agent.

The "United Airlines-Microsoft venture" is an indication of the desire of vendors to by-pass travel agents. Travel agencies have to acknowledge the threat of vendors automating the travel agent's function and doing away with travel agents. To stay alive, travel agencies will have to adopt and exploit enabling technologies faster than vendor alliances and travellers do.

2.4.3 Vendors form Alliances and use Intelligent Agents to bypass Travel Agents

The acrimonious relationship between travel agents and vendors shows that vendors view travel agents as a necessary evil. Given the opportunity, vendors would get rid of travel agents. Presumably, eliminating travel agents altogether would yield great profits. Vendors of travel related services are already forming alliances. For example, an airline, a car rental agency and a hotel chain will bundle their services and provide a traveller with a group discount on the package. The car rental agency will provide a customer with frequent flier miles on the participating airline to motivate the customer to fly with the airline.

In this model, it is the vendors who own the Intelligent Agents. The traveller in search of services locates a site from which the Intelligent Agent is triggered. The traveller provides the criteria for the search and sends the agent on its way. The agent puts together a package from the alliance and returns the results to the traveller. A percentage of the transaction costs will cover the cost of the Intelligent Agent.

Unless a traveller owns an Intelligent Agent, this model limits the choice of services to the alliance that owns the agent employed. A traveller would have to deploy a personal Intelligent Agent to a number of alliances in order to choose among different alliances.

2.5 Future Evolution

2.5.1 The drawbacks of software travel agents[1]

Does this mean that people have to give up the traditional way of booking business and holiday trips and use the World Wide Web instead? The truth is that existing software agents are not as effective and that people are not ready to risk it.

Time Consumption

When customers know exactly what they want they can make a phone call to their favourite travel agency and book in a matter of minutes. But if they know exactly what they want they can also connect to a relevant web site, or even connect to several web sites and compare ticket prices. The big drawback, though, is that at peak hours, (usually at noon), Internet access is really slow and it can get quite annoying.

Reliability

When customers book through human travel agents they usually expect (and receive) good service. They rely on the agents and can blame them if something goes wrong with the booking. When booking through software agents, success depends on the customer. If anything goes wrong, the customer cannot hold anyone else responsible but himself.

[1] based upon [MICHAELIDE et al. 1997]

Support

Uncertain customers can seek the advice of their favourite travel agency regarding where they should go according to the money they have available and their personal desires. Uncertain customers that are seeking advice from software agents on the Web have no hope in finding any help (at least for the next few years). Maybe that is the main reason why travel agencies are still necessary....

2.5.2 Is there a future for travel agencies?

In the landscape of the software travel agents, travel agencies are probably the most endangered organisations, since their job is increasingly seen as being replaceable by technology. Today, travel agencies play multiple roles: information brokers to pass information from product suppliers to customers, transaction processors to print tickets or forward money, and advisors to provide value-added information to their customers, assisting them in their choice of specific products and destinations. The first two of these roles will increasingly be played by technology, going directly to the customers to provide them with information and process their transaction needs. Therefore, agencies will have to focus on the third role, and differentiate this role according to their target market.

Indeed, one may assume that, provided the traveller knows what he/she wants, filling up a form (with the destination, date, class,...) and submitting it is a piece of cake. But does the customer always know what he/she wants when he/she enters a travel agency? Often, customers, usually looking for holiday packages, depend on their travel agent to show them what is available according to their taste and budget. Unfortunately, nowadays, there exists no software travel agents that are more intelligent, i.e. that are able to interact with the customer on a more personal basis. Clearly, the role of a travel agency as intermediary can remain important for the industry.

3. Characteristics of our application

Now that we know what an agent is and what the characteristics of a travel system are, we can define what kind of agent we built. In this chapter, we will see how the concepts related with agents are present in our application.

3.1 Brief description of the application

We built a system of agents composed of two kinds of agents : **travel agents** and **company agents**. Travel agents' goal is to buy the ticket matching the best the traveller's preferences. The airline companies' agents give all the information the travel agents need to choose a ticket. We go deeper into the description of our application in chapters 6 and 7.

Since our application is a very simple prototype that is limited to a system of one (or more) travel agent (s) and three vendors (three airline companies; we did not deal with car rental, hotel booking or other services), we cannot classify it in one of the three scenarios. Our small system could be part of a bigger system that embodies one of the three scenarios presented in chapter 2.

3.2 Tools used to build the application

To implement our travel agent, we had to make some choices. What programming language would we use? Were there tools at our disposition, for example to treat non-monotonic logic? What had to be the features of our agent? What kind of architecture would fit the best? Did we need to use an Agent Communication Language?

We first had to choose a programming language. Obviously, an agent-oriented language can be used for developing and running intelligent software agents. Before really starting the implementation of our agent, we had the opportunity to take part in seminaries introducing the **Jack**[1] language. It seemed to be adequate to the type of agent we wanted to develop. Jack is an agent-oriented language developed by Agent Oriented Pty. Ltd[2]. Chapter 4 tells us more about Jack.

For implementing the modules that treat non-monotonic logic (Belief Revision and Default Logic), we preferred to make use of available generic tools instead of ad hoc programs that we could have developed. The tools we chose are **Vader** (a theorem prover), **Saten** (for Belief Revision) and **Hades** (for Default Logic). They all were developed at the university of Newcastle, Australia, by the CIN project[3]. Chapter 5 gives more details about these tools.

[1] http://www.software-agent.com
[2] http://www.agent-software.com/home.html
[3] http://infosystems.newcastle.edu.au/maryanne/projects/CIN.html

3.3 Features of the agent system[1]

3.3.1 Properties and typology of the agents

We take a closer look here at the agent system we developed. We mainly focus on the travel agent as the company agent has a very simple architecture and don't have complex reasoning methods.

Our travel agent exhibits the three primal attributes of an agent: It is **autonomous**. Once it is created, it doesn't need human guidance. It is **reactive** because it reacts in response to its environment. For example, it triggers off some actions to respond to a message from another agent or to a specific behaviour of the user. On the other hand, it is **pro-active**. Indeed, it acts in a goal-oriented way thanks to the structure of plans offered by Jack. When our agent has a goal, it is able to decompose it into sub-goals in order to achieve it.

Our agent is also **adaptive**, i.e. it behaves differently depending on the environmental conditions. There are tracks of **memory** (for example, it remembers all what it has been asked since its creation) and **reactivity** to environmental changes. But the agent doesn't have the capacity of **learning** because it doesn't really adapt its behaviour to the one of the traveller.

Furthermore, it is capable to **co-operate** with the user and with other agents. The co-operation with the user is embodied by the interface of the agent. The user has to specify the task he/she wants the agent to achieve. Once the agent has performed its task, it warns the user of the result. The user can then ask another task or modify his/her initial request. Our travel agent also co-operates with other agents (the company agents) that will help it to achieve its goal, mainly by providing information. In order to do so, an agent communication language is necessary. The communication language our agents use is integrated in Java, on which Jack rests.

Finally, the travel agent has the properties of **veracity** (it doesn't knowingly communicate false information), **benevolence** (it always tries to do what is asked of it because it doesn't have conflicting goals) and **rationality** (it acts in a way that is optimal for achieving its goals). But it is neither **introspective** (it can't examine and self-reflect its own thoughts, ideas, plans,...) nor **mobile** (it is not able to move to another system trough an electronic network to access remote resources or to meet other agents). Mobility is a very interesting property but it was not necessary for our agent. Moreover, Jack is not adapted to the development of that kind of agent.

Now that we have examined the properties our travel agent possesses, we can conclude that it ranges in the category of **collaborative agents**. It can be a personal agent provided by a special vendor and owned by the user or by the vendors but it can also be the agent of a travel agency.

[1] this section should be read in parallel with the more theoretical second chapter of part I

3.3.2 Agent architecture and communication language

We built the internal architectures of our agents following the BDI architecture. It is the most accepted architecture and what's more, the Jack architecture is based on the BDI architecture. In chapter 6, we will analyse the BDI features of the agents and then build the architecture following the Jack requirements.

Concerning the architecture of the agent system, we did not use standards developed by the FIPA or OMG. Our application is targeted at the internal architecture of the agent. As a consequence, there are no general services such as a naming service in our systems. Each agent knows each other agent of the MAS.

However, it is not impossible to extend the actual prototype to a more sophisticated system using FIPA or OMG standards. Java facilities for distributed objects also offer a strong basis to build distributed systems. Standard architectures are chosen for wide applications used on global networks such as the Internet. Specific architectures are chosen for more specific applications limited to a small group of users.

Concerning the communication, we did not use standard ACL in our application. Jack offers a good communication language and it is easy to use. Standard ACLs can be really useful for huge applications but the use of such languages is more complicated.

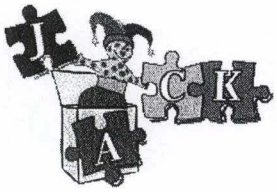
In the case of large travel systems, it seems that the application will be used by a wide variety of users. Therefore, standards are appropriated so that each airline company can build its own agent following some conventions. Consequently, a wide variety of travel agents can be built using the services of the airline agent and offering information for travellers.

The use of ACL standards is helpful when used with MAS standards. It is not yet the case with our application but, once again, it is not impossible to extend it using KQML for example. Furthermore, if we want to integrate our agents in a wider application (e.g. the Internet), standards will be needed.

3.3.3 Artificial intelligence

One of the goals of our dissertation is to incorporate complex AI-based reasoning methods in our agent. Hence, we endowed our agent with non-monotonic reasoning. We have decided to use Default Logic for completing missing information about the tickets proposed by the airline companies and Belief Revision for managing the preferences of the traveller. We will explain more in detail how we use these two types of non-monotonic logic in section 7.2.

4. An agent oriented language (Jack)



In the present chapter we describe Jack, the language we used to build our application. Jack is a commercial product developed by Agent Oriented Software (AOS) based in Melbourne, Australia. We first see that it is based on the BDI architecture and Java. Next, we describe the components of a Jack agent and see the benefits of the Jack language.

4.1 Introduction[1]

The goals set by the Jack[2] designers were to provide developers with a robust, stable, light-weight product, to satisfy a variety of practical application needs, to ease technology transfer from research to industry and to enable further applied research.

Because of most organisations already posses and depend upon large legacy software systems, the Jack Agent has been designed mainly for use as components of larger environments. Consequently, an agent must co-exist and be visible as simply another object by non-agent software. Conversely, a Jack programmer must be allowed to easily access any other component of a system.

Jack agents are not bound to any specific agent communications language but Jack provides a native communication infrastructure. Moreover, object-oriented Middle-Ware such as CORBA (developed by the OMG, see section 2.4.2 of part I) can be combined with Jack to solve some communication issues.

Jack has two main features. First, the Jack agent architecture is inspired from the *Belief-Desire-Intention (BDI) model* (see section 2.4.1 of part I). Secondly, Jack is a *Java-based language*. That means that Jack users can use all the tools supplied by Java to build a Jack agent system.

4.2 Jack and the BDI model

A Jack agent is a "BDI Agent" (cf. section 2.4.1 of part I). Such agents have *beliefs* about the world and *desires* to satisfy, driving it to form *intentions* to act. An intention is a commitment to perform a plan. Beliefs, desires and intentions are called the mental attitudes (or mental states) of an agent.

The abstract BDI architecture has been implemented in a number of systems. Of these, two are of particular relevance to Jack since they represent its immediate predecessors. The first generation is typified by the Procedural Reasoning Systems, developed by the SRI

[1] this chapter is based upon [BUSETTA et al. 1999], [COBURN 2000]
[2] http://www.software-agent.com

International in the mid '80s. The second generation systems is dMARS, built in the mid '90s by the Australian Artificial Intelligence Institute (AAII) in Melbourne. dMARS has been used as the development platform for a number of technology demonstrator applications, including simulations of tactical decision making in air operations and air traffic management.

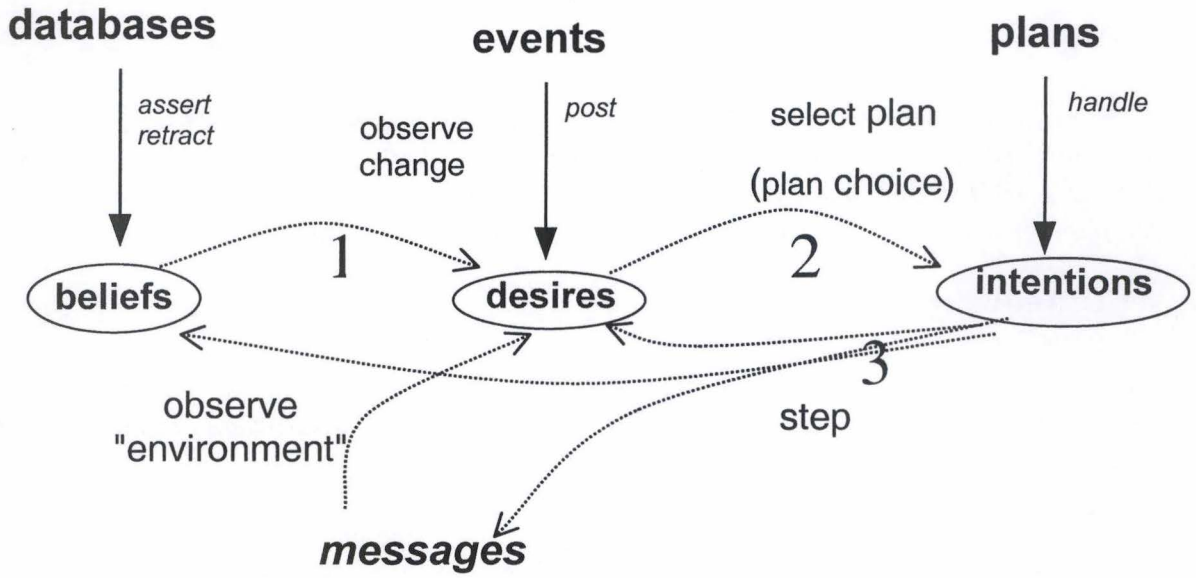


Figure 10 : Jack BDI Execution

A Jack agent has three main types of components : the databases, the plans and the events. As shown on Figure 10, these Jack components are in close relation with the BDI components (i.e. the beliefs, desires, and intentions). The agent's *beliefs* can be asserted or retracted from the databases, its *desires* are posted by the events and its *intentions* are handled by plans.

Clearly, there is a similarity between the Jack BDI architecture and the BDI architecture seen in section 2.4.2 of part I. Figure 10 describes the Jack BDI execution. First, belief changes generate new desires. This means that changes in the databases generate the creation of new events (1). Then, the agent has to choose intentions to satisfy the desires. The selection of appropriate plans determines the new set of intentions (2). The selected plans are then executed. The plan's instructions (step intention) will then generate new desires (i.e. generate an event) and change the beliefs. The plan's instructions can also send messages to other agents. These messages can change the desires of the agent that receives the message (3).

4.3 Jack : a Java-based Language

A Jack programmer uses Java statements within the specific components of a Jack agent. Jack currently consists of three main extensions to Java. The first is a set of syntactical additions to its host language, which can be divided as follows :

- A small number of keywords for the identification of the main components of a Jack agent. The keywords describe components such as plans, events and agents.
- A set of statements for the declaration of attributes and other characteristics of the components. This is the information contained in the beliefs or carried by the events.
- A set of statements for the definition of static relationships, i.e. which plan can be adopted to react to a certain event.

- A set of statements for the manipulation of an agent's states. This comprises additions of new goals, changes of beliefs and interaction with other agents.

The second extension to Java is a compiler that converts the syntactic additions described above into pure Java.

Finally, the third extension is a set of classes (called the kernel) that provides the required run-time support to the generated code. This includes :

- the automatic management of concurrency among tasks being pursued in parallel (the intentions in the BDI terminology).
- the default behaviour of the agent in reaction to events, failure of actions and tasks, and so on.
- a native light-weight, high performance communications infrastructure for multi-agent applications. Note that a different communications infrastructure can be supplied by overriding the appropriate run-time methods.

In summary, the Jack Agent Language (JAL) is a third generation language that follows the standard Java/object-oriented paradigm, now commonly accepted and understood by software developers.

The choice to extend Java to create Jack language is judicious. All the advantages of Java are incorporated automatically in Jack. Indeed, Jack allows access to all Java capabilities, including multiple threads, possibility running on multiple CPUs, platform independent GUIs and third party libraries. For example, Jack programmers can build distributed object systems using Java functionalities. Moreover, Jack allows easy integration using standard distributed-object Middle-Ware infrastructure, such as CORBA, RMI, or DCOM. Another important Java feature that Jack inherits is the portability. Jack agents are capable of running on any system with Java, from mid-range laptops to high-end multi-CPU Enterprise servers.

We have seen in section 2.7 of the first part that an agent language is often based on object-oriented languages but that they also need to offer some facilities to use Artificial Intelligence techniques. This is why Jack supports logical variables and cursors for the convenience of AI programmers. These are particularly helpful when querying the state of an agent's beliefs. Their semantics are mid-way between logic languages and the SQL language. However, generally speaking, Jack is much more object-oriented than AI-oriented.

4.4 Jack programming concepts

The Jack Agent Language (JAL) can be categorised as follows : the JAL classes (types), the JAL declarations (#-declarations) and the Jack reasoning method statements (@-statements).

The classes define functional units within Jack. These functional units are implemented as Java classes, with their agent-oriented properties embedded within the class as private methods. There are five pre-defined classes :

- *Agent* which models the main entities in Jack
- *Event* which models occurrences of messages to which these agents must be able to respond. Events may arise externally from messages from other agents, or internally as a consequence of one of the agent's own actions or in response to one of its internal goals.
- *Plan* which models procedural descriptions of what an agent does to handle a given event.

- *Database* which models an agent's beliefs. Databases represent an agent's belief as first order relational tuples.
- *Capability* that models a kind of mini-agent included in an agent. The capabilities provide a way to structure the agent by providing an independent entity for a particular function. For example, an agent may have the capability to compute a factorial number. Each capability has his own plans, events and databases. The capability construct allows the functional components to be reused.

Figure 11 represents the five classes and their mutual relationships. An agent has events, plans, databases and capabilities. A capability is a kind of mini-agent and, therefore, can have capabilities, events and databases. Changes in databases and instructions in plans may generate events. A plan's instruction uses the knowledge contained in the databases.

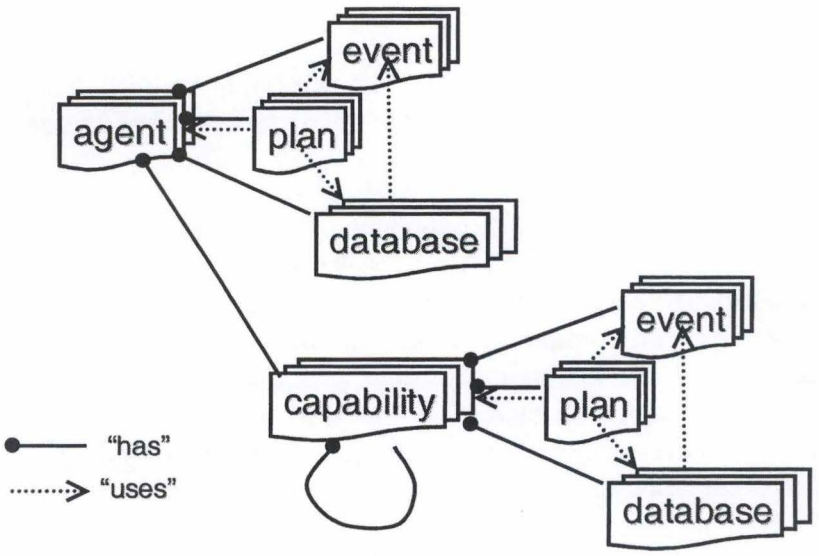


Figure 11 : Jack programming concepts

Jack files have the following extensions : *x.agent* (for the agent classes), *x.plan* (for the plan classes), *x.event* (for the event classes), *x.db* (for the database classes), or *x.cap* (for the capability classes). As seen in section 4.3, Jack provides a compiler that transforms these Jack files into Java files (*x.java*). To run the agent software, the Java compiler has to transform the *x.java* files into *x.class* files that can be interpreted by the Java virtual machine.

Declarations are used to specify relationships between classes in a Jack program.
For example :

- *#uses plan* - is used in agent definitions to specify that an agent includes this plan in its set of available plans.
- *#handles event* - is used in agent definitions and plan definitions to identify the event that an event or plan handles (one plan handles one and only one event).

Reasoning statements are JAL specific statements that can only appear in reasoning methods. They describe actions that the agent can perform and that are not covered by normal Java.

For example :

- *@send* - sends a message event to another agent.
- *@wait_for* - identifies a condition for which the agent should wait until it becomes true.

Of course, each Jack Agent Language class also offers a number of public methods. For example, there are three database methods :

- *assert()* - adds new tuples to an agent's private database relation, or modifies its existing tuples by supplying updated information.
- *retract()* - removes tuples from an agent's database relation.
- *public int nFacts()* - returns the number of facts (tuples) that are currently held in a given database relation.

4.5 Benefits of Jack

The approach taken by Jack has a number of advantages. The adoption of Java guarantees a widely available execution environment. Moreover, it is probable that an increasing number of software components using Java will be available in the next few years.

However, the adoption of an imperative language such as Java means losing some of the expressive power offered by logic or functional languages. This is partially compensated with the advantages offered by the intrinsic characteristics of Java, and, therefore, by Jack. For example, Jack has several interesting aspects for developing a complex distributed system. Jack has also the advantage of being accessible to a large community of engineers trained in object-oriented programming.

In summary, Jack constitutes an elegant marriage between the vision of agent research and the needs of software engineering, bringing the power of agent technology to and enriching the host language Java.

4.6 Evolution

For developing our application, we used the Jack Intelligent Agent v1.3. In the mean time, a new version has appeared (v2.0). Jack Intelligent Agents v2.0 include all the agent programming concepts of Jack v1.3, and has obviously some additions. We will discuss shortly the most important ones. A first important improvement in the new version is a graphical development tool for Jack agents. Secondly, Jack v2,0 provides an helpful documentation in *javadoc* style. And thirdly, Jack v2.0 provides additional utility classes.

The graphical development tool is entirely written in Java. It can be viewed as a toolkit that facilitates the construction of the component parts of an Agent - Capabilities, Plans, Events, and Databases. These parts can then be used to build the agent you need for a given application. One of the advantages is that the Jack Development Environment makes it easy to reuse code, especially via the Capabilities mechanism.

However, the changes in the new version are not that important because the philosophy to build Jack agents remains the same. Nevertheless, it is interesting to know that the product evolves.

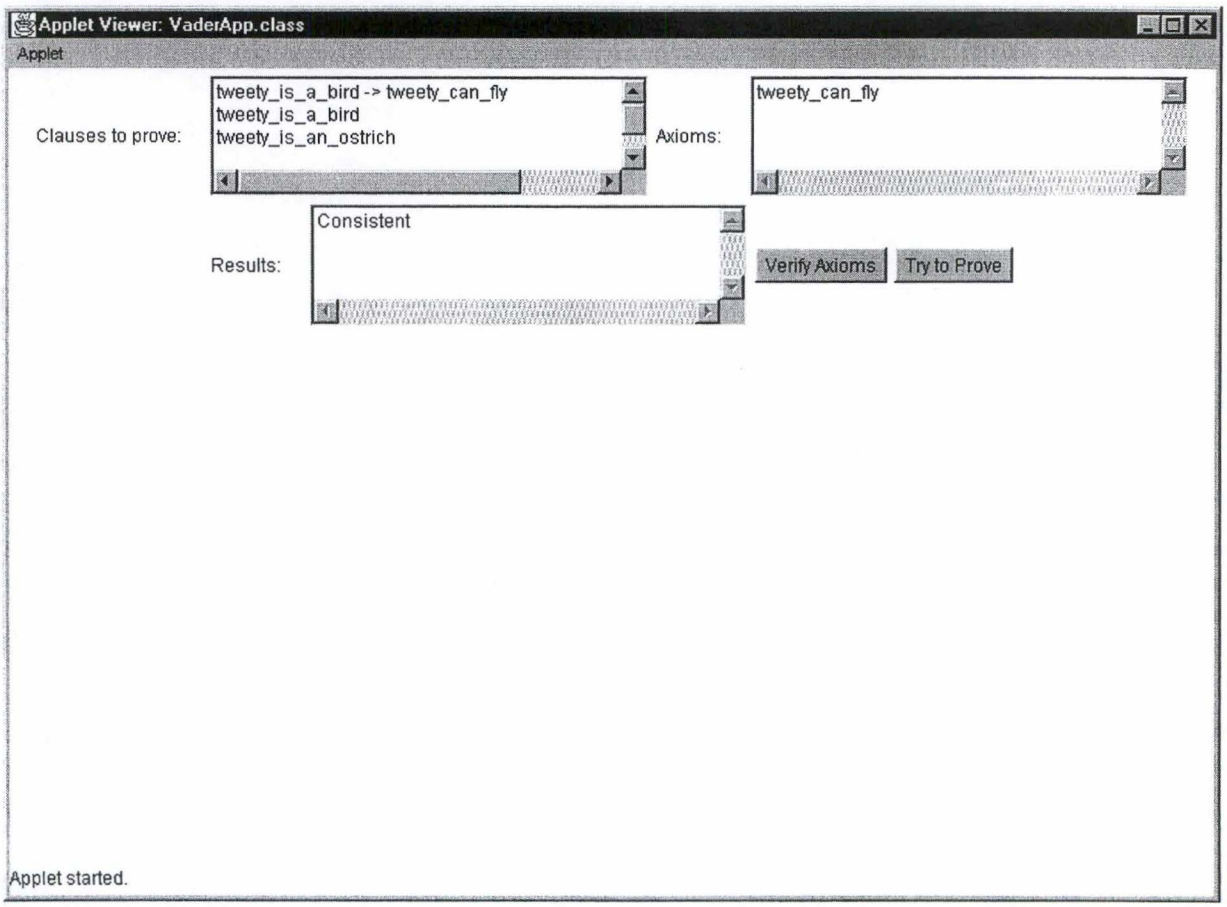
5. AI tools : Vader, Hades, and Saten

The UNIVERSITY of NEWCASTLE AUSTRALIA

To develop our applications, we needed tools that deal with non-monotonic logic. We used Saten and Hades. Saten treats Belief Revision whereas Hades treats Default Logic. Both Saten and Hades were developed in Java, by the CIN project at the Business Technology Research Laboratory of the University of Newcastle, Australia. They use an object-oriented web-based first-order theorem prover, Vader.

5.1 VADER

Vader[1] was implemented by the CIN project in 1997. It is believed to be the first theorem prover fully implemented in 100% pure Java. It was upgraded to first-order logic in 1998.

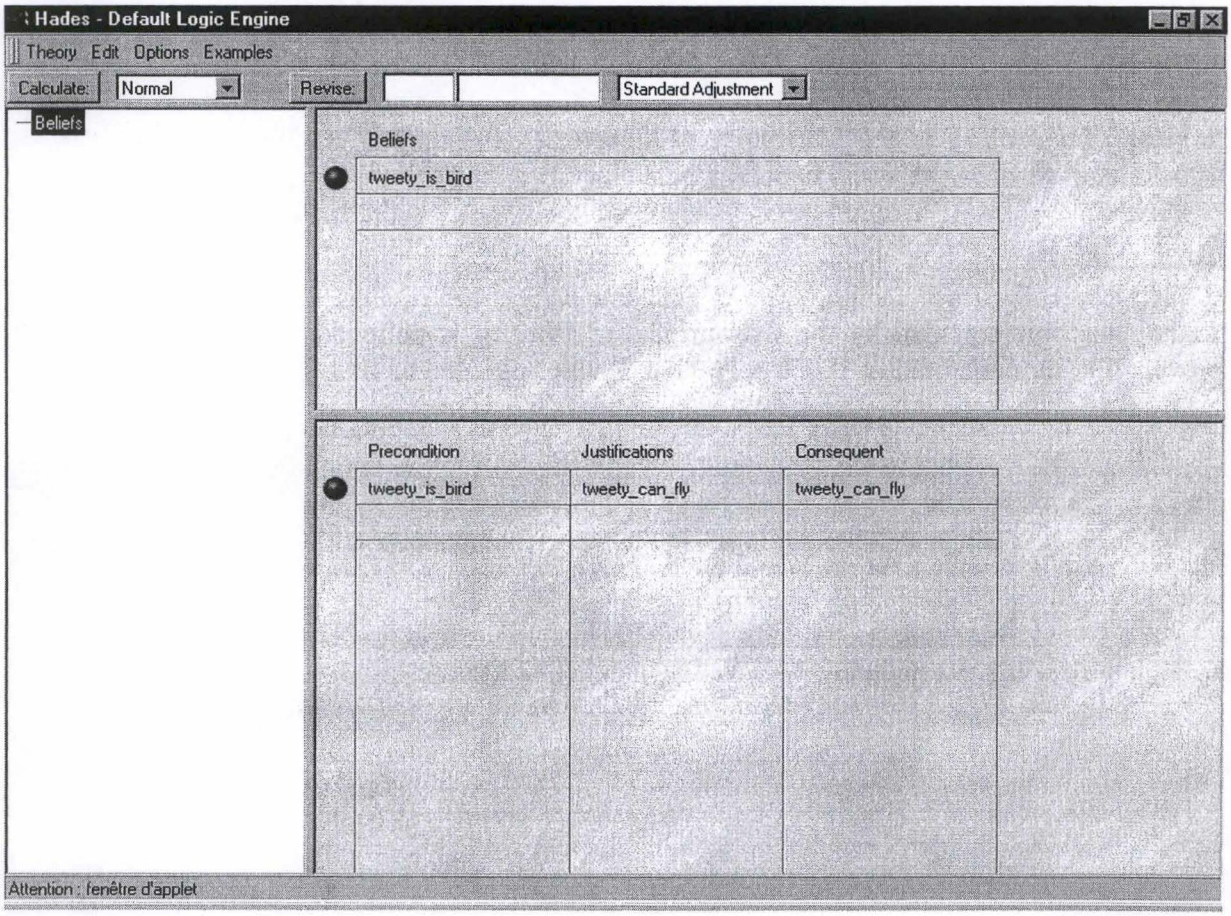


You can either ask Vader to verify whether the clauses to prove are consistent with the axioms (he will then answer 'consistent' or 'inconsistent') or ask it to try to prove the clauses with the axioms (he will then answer 'proven' or 'not proven').

[1] http://cafe.newcastle.edu.au/vader/

5.2 HADES

Hades[1] was developed as an intelligent object-oriented prototype system for Default Logic with Belief Revision capabilities (by using Saten).



The interface is divided in two tables, which the user has to fill in : the beliefs (on which one can use the button 'Revise' in the same way as in Saten, see section 5.3) and the defaults. The underlying data structure is a *default theory*. It is composed of an array of *strings* (*in*) for the beliefs (or facts), an array of *default* objects (*defaults*) for the defaults and an array of arrays of *String* (*extensions*) for the extensions.

The class *default* is composed of a *pre* (a *string* representing the precondition or prerequisite), a *justs* (an array of *strings* representing the justifications) and a *cons* (a *string* representing the consequent).

The main methods of the *default theory* are :

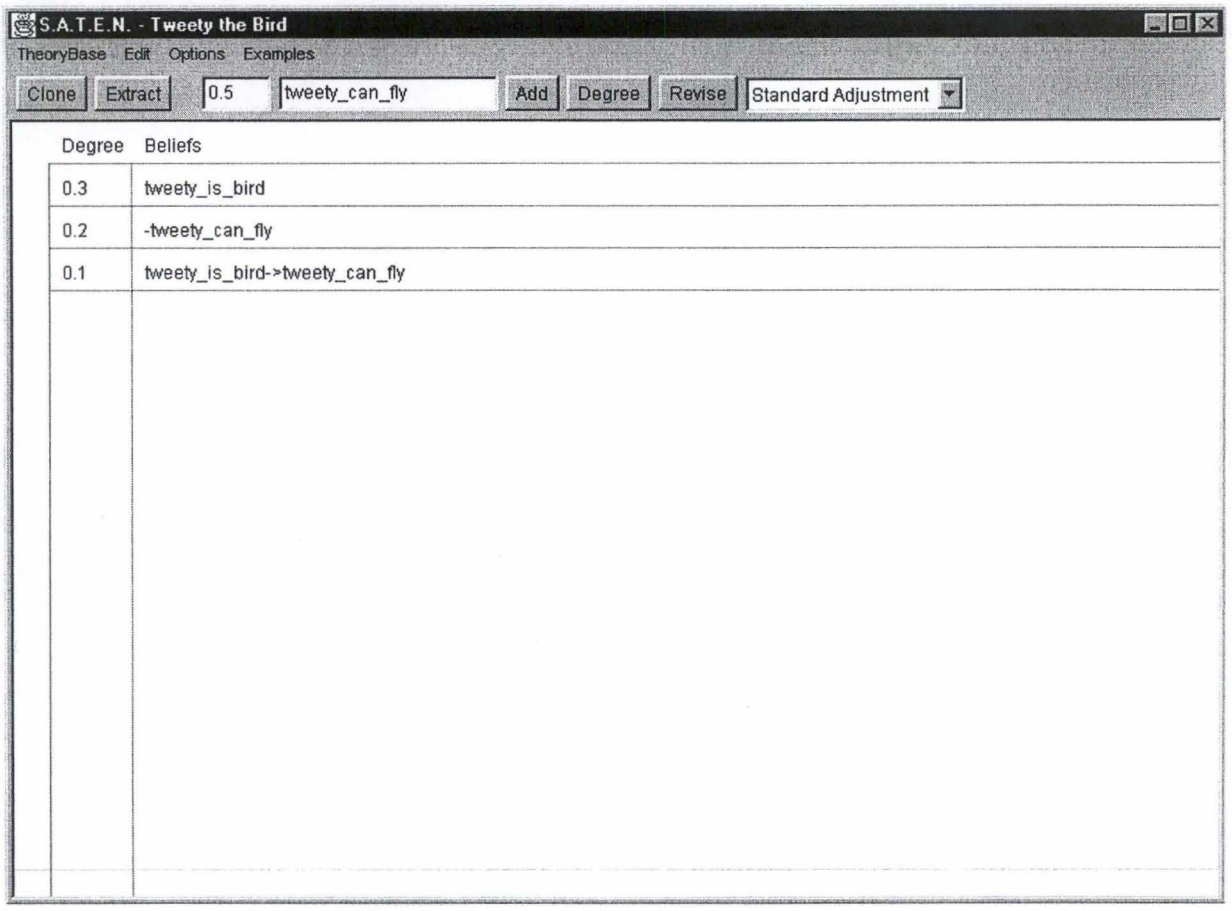
- *addDefault(Default)* - inserts a new default in the DefaultTheory. The parameter is the default to be inserted.
- *addFact(String)* - inserts a new fact in the DefaultTheory. The parameter is the fact to be inserted.
- *generate()* - generates all the extensions and returns them in an array of arrays of string (*extensions*).

[1] http://infosystems.newcastle.edu.au/webworld/Hades/

Obviously, we did not make use of Saten and Hades the way one does on the Web site. We did not use the interface, we only used the engines underneath it. We utilised the extraction engine of Saten and the generation engine of Hades.

5.3 SATEN

Saten[1] was also developed in 1997. It is an intelligent object-oriented prototype system for theory extraction and revision.



The interface is mainly composed of a table that the user has to fill in with the beliefs and their degree. In order to insert a new belief in the table (which represents the beliefs set), one can either use the 'Add' button or the 'Revise' button. Pushing that button means revising the beliefs set with the new belief, following the chosen strategy (Standard Adjustment, Maxi-Adjustment,...). If the 'Add' button is used, the newly formed beliefs set can be inconsistent. One can then use the 'Extract' button to extract a consistent theory from it. Finally, one can use the 'Degree' button to ask for the degree the specified belief would have in the beliefs set.

Saten is capable of :

- Theory extraction
- Information integration/fusion
- Iterated revision
- Calculating Spohnian reasons

[1] see also http://infosystems.newcastle.edu.au/webworld/Saten/

- Non-monotonic reasoning
- Possibilistic reasoning
- Hypothetical reasoning

In the application we developed, we only made use of two of these capabilities : theory extraction (i.e. extraction of a consistent subset of formulas from a set of formulas that can be inconsistent) and iterated revision. These are the two main functions of Saten. Both of them are performed on a ranking that the user is expected to provide. Extraction involves the recovery of a consistent ranking from an inconsistent ranking. Several strategies are available for the process of extraction and revision. These strategies are based on the different types of transmutations :

- Standard adjustment
- Maxi-adjustment
- Hybrid adjustment
- Global adjustment
- Linear adjustment

In our application, we only use maxi-adjustment.

The data structure at the basis of Saten is the *theory base*. It is an array of *strings* (the beliefs) coupled with an array of *doubles* (the ranking of the beliefs). These arrays have the same size because there is a one-to-one correspondence between them : to a belief (represented by an element of the array of *strings*) corresponds a rank (represented by an element of the array of *doubles*).

The main methods of the *theory base* are :

- *addBelief(String, double)* - inserts a new belief in the TheoryBase. The parameters are the belief to be inserted and the entrenchment at which to insert it.
- *getEnt(String)* - gets the ranking associated with a given belief in the TheoryBase. The parameter is the belief of which we search the ranking.
- *contains(String)* - determines if a given belief is in the TheoryBase. The parameter is the belief to be searched for.
- *movBelief(String, double)* - moves a belief within this TheoryBase. The parameters are the belief to be moved and the new entrenchment for this belief.
- *remBelief(String)* - removes any copy of a given belief from the TheoryBase. The parameter is the belief to be removed.

To make the revision of the TheoryBase, a different object that depends on the strategy is used. It can be the object *Adjust* if the strategy is Standard Adjustment, the object *MaxiAdj* if the strategy is Maxi-Adjustment, *HybridAdj* if the strategy is Hybrid Adjustment, ... All these objects are instances of classes that extend the class *Extractor*. This constitutes the base class for the Theory Extraction Engine hierarchy. It allows for Theory Extraction (and hence Belief Revision) to be applied to TheoryBases. Its main methods are extractTheory (TheoryBase) and revise (String, double).

extractTheory extracts a consistent theory from the current TheoryBase. It uses a clausal form and a theorem prover (Vader) to get a consistent theory, and then decides which non-clause-form beliefs are provable from this base and at what entrenchment. This becomes the new TheoryBase.

revise uses the extraction engine to perform a Belief Revision on the current TheoryBase. The parameters are the belief to be revised and the entrenchment at which to insert this new belief. Basically, it consists of adding a belief to the TheoryBase and using the method extractTheory on the new TheoryBase.

Other important methods of the class *Extractor* are *MoveDown* and *MoveUp*. It moves the nominated belief down/up to a new entrenchment, moving other beliefs as necessary to maintain the entrenchment property.

6. Architectures

In chapter four, we have seen what the components of a Jack agent are. In this chapter, we analyse the BDI-features of each agent that composes the system so that we can build a Jack architecture for each agent. We describe first the interaction between the agents in the system. Afterwards we analyse the characteristics of each agent in more detail.

6.1 External architecture

The travel reservation system is composed of two kinds of agents. "TravelAgents" that buy tickets for travellers and "CompanyAgents" that sell tickets for airline companies. The goal of a TravelAgent is to find the "best ticket" corresponding to the traveller's preferences.

The TravelAgent first asks the traveller for his or her preferences. The traveller has to choose his/her degree of preference for each criterion (measured on a scale that goes from 0 to 10). The preferences concern the price of the ticket, the duration of the flight, having a smoker seat or not, having a personal television, etc. Of course, the traveller has also to specify the date on which he/she wishes to leave, and his/her place of departure and his/her destination.

The task of the TravelAgent is to find the ticket that matches the best the preferences expressed by the traveller. To this end, the TravelAgent has to communicate with CompanyAgents. A TravelAgent asks a CompanyAgent information about seats for a flight linking a specific place of departure to a specific destination. As a reply, CompanyAgents provide all the information relating to the tickets the airline company possesses.

Once the TravelAgent has found the "best ticket" for the traveller, it asks the CompanyAgent if there are vacant seats. If this is the case, the agent books the ticket. If this is not the case, the agent queues up, waiting for a possible cancellation and another search will be spawn to find the "second" best ticket. This process will be executed until an available seat is found.

Two causes can modify the selection of the "best ticket". First, each time there is a change in a ticket's characteristics, the CompanyAgent sends the changes to the TravelAgents that had asked information about this kind of ticket. In this way, the TravelAgent can perhaps find a new "best ticket". Second, travellers can also change their preferences during the reservation process and thus change the "best ticket" corresponding to their preferences.

The CompanyAgent informs the TravelAgent if former queued tickets are now available. By doing so, the TravelAgent can cancel already booked tickets. The reservation process ends some days before departure when time has come to pay the "best tickets".

As shown on figure 11, our prototype system runs with three CompanyAgents (Virgin, KLM and Sabena) and two TravelAgents. In our application, each agent is identified by a name and it suffices to know the agent's name to be able to send a message to an agent.

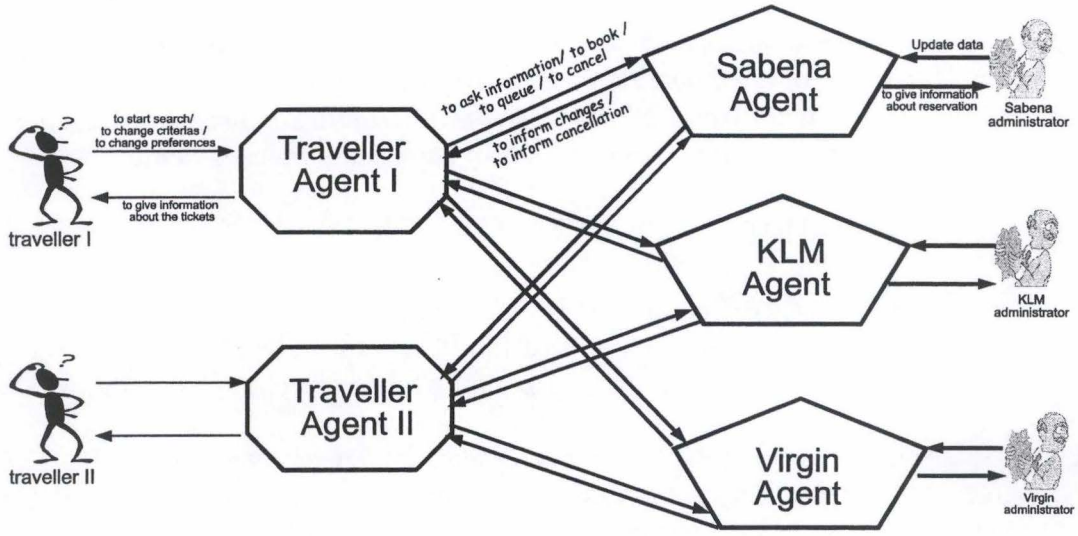


Figure 12 : Prototype of a travel reservation system

6.2 TravelAgent

6.2.1 Features of the TravelAgent

The most complex agent is the TravelAgent. To build the agent, we have to identify what kinds of reasoning methods will be used by the agent, which external events drive the agent, which goals the agent can set for itself, which beliefs the agent has and finally, which functionalities have to be provided by the agent.

Reasoning methods

The particularity of our application is that the TravelAgents use non-monotonic reasoning. They organise the preferences of the traveller and the information they know about the possible tickets in the form of a Theory Base (the data structure used in Saten to represent the belief set) and use Belief Revision to single out the ticket that matches the best the preferences of the traveller.

Default Logic is also used to deal with incomplete data coming from the CompanyAgents. They can forget, for instance, to tell about the presence of smoker seats on a flight, or maybe they just do not tell it because it seems to them that it is implicit.

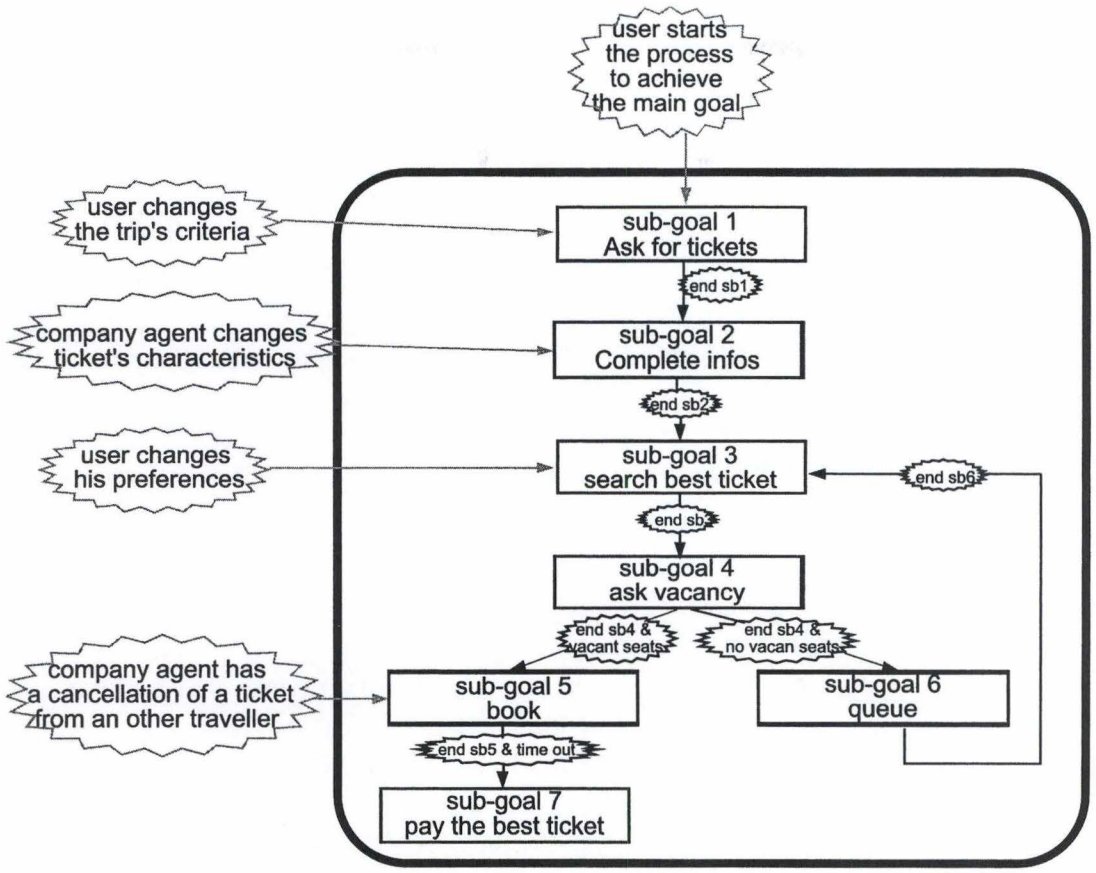


Figure 13 : External events and Sub-goals of the TravelAgent

External events

As shown on Figure 13, the TravelAgent can receive external events from the traveller and from CompanyAgents. The traveller can induce two kinds of events by modifying the criteria of the trip (e.g. the destination) or by changing his or her preferences (e.g. he/she changes his/her mind about the fact of being served champagne). The CompanyAgent can also induce two kinds of events by informing the TravelAgent about changes of a ticket's characteristics (e.g. the disappearing of smoker seats on a flight) or about a ticket's cancellation from other travellers.

Goals of the TravelAgent

The main goal of a TravelAgent is to find and to book the "best ticket" according to the preferences of a traveller. The main goal can, of course, be divided in sub-goals. The successive achievements of sub-goals characterise the progress of the main goal.

We can divide the main goal in seven sub-goals:
First, once the search is started, the agent has to ask the CompanyAgents which tickets match with the traveller's desire (sub-goal 1). Secondly, the agent has to complete the information provided by the CompanyAgents using the Default Logic (sub-goal 2). Thirdly, the agent has to single out the ticket matching the best with the traveller's preferences thanks to a selection system based upon the Belief Revision technique (sub-goal 3). Fourth, the agent asks the CompanyAgent (that provides the "best ticket") if there are vacant seats for the "best ticket"

(sub-goal 4). If this is the case, the agent books the ticket (sub-goal 5). If not, the agent queues up, waiting for a possible cancellation (sub-goal 6) and another search is spawn to find the "second" best ticket (go back to sub-goal 3). Finally, when the laps of time to pay the ticket is over, the agent pays and the process ends (sub-goal 7).

The four external events may restart the process from a sub-goal. If the traveller changes the trip's criteria (e.g. the place of departure) the whole process has to be started all over again. If an agent company changes a ticket's characteristics, the agent has to recalculate the default characteristics. If a traveller changes his or her preferences, the agent has to recalculate the "best ticket". Finally, if a company has a cancellation for a ticket for which the TravelAgent queues up, the agent books the ticket.

Beliefs

The beliefs of a TravelAgent concern all the data that an agent has to know to achieve each sub-goal. We will look more into detail at each sub-goal below.

- To ask for tickets (i.e. sub-goal 1), the agent has to know the trip's criteria and the name of each CompanyAgent, so that they can communicate.
- To complete missing information (i.e. sub-goal 2), the agent has to know the information on the tickets sent by the CompanyAgents.
- To find the "best ticket" (i.e. sub-goal 3), the agent has to know all the characteristics of every ticket.
- To ask if there are vacant seats for the "best ticket" (i.e. sub-goal 4), the agent has to know which is the "best ticket" and it also has to know the name of the CompanyAgent that provides this "best ticket".
- To book the "best ticket" or to queue (i.e. sub-goal 5-6), the agent has to know which is the "best ticket" and if there are vacant seats left. The name of the CompanyAgent is once again needed to book the "best ticket" or to queue.
- Finally, to achieve sub-goal seven, i.e. paying the last "best ticket", the agent has to know which is the "best ticket" having at least one available seat and it needs to know when it's time to close the process.

Functionalities

To implement the agent, we have to define all the functionalities the agent has to provide. These functionalities define which functions are needed to achieve the sub-goals. Now that we have decomposed the goal into sub-goals, it is easier to specify which functions are needed to achieve the main goal. We have defined seven groups of functionalities for the TravelAgent.

First, the agent has to be able to communicate with the traveller and with the CompanyAgent since the agent has to be informed when an external event occurs. As a consequence, two kinds of functionalities are needed: one that regroups functions that permits communication with the traveller and another that regroups functions that allow to communicate with other agents in order to ask if there are vacant seats or in order to receive information about tickets for example.

The agent has also to be able to use non-monotonic logic. This means that the agent uses Default Logic and Belief Revision. Therefore three groups of functionalities are needed: one

to use Saten (Belief Revision), another to use Hades (Default Logic) and yet another to transform the data into logic expressionss adapted to the belief revision treatment.

Then, Functions allowing to book and queue are regrouped in another functionality.

Finally, another faculty that can be useful as well, is the faculty that allows memorisation of all the research executed by an agent so that the agent never executes twice the same process. Thus, each time the traveller modifies the trip's criteria, the result is memorised with all the parameters.

6.2.2 Jack architecture

Now that we have analysed the main features of the TravelAgent, we can begin distinguishing between the different components of a Jack agent. As we have already mentioned (chapter 4), a Jack agent has five kinds of components : the agent itself, the capabilities, the databases, the events and the plans. On Figure 14, we can see how these capabilities are co-ordinated to achieve the sub-goals.

Capabilities

A capability will be built for each group of functions except for the communication between the agent and the traveller. A Graphical User Interface (GUI) will cover this functionality.

Therefore, a TravelAgent has five capabilities and an interface :

- *Communication* regroups all the functionalities to communicate with a CompanyAgent.
- *DefCap* uses Hades to complete missing information.
- *TreatDataCap* takes care of the form of the formulas that will be used in the Br capability.
- *Br* uses Saten to find the "best ticket" by using Belief Revision.
- *BookingCap* manages all the queued and booked tickets.
- *History* remembers all the previous research made by the agent.

Plans

Each plan of the capabilities will correspond to each function provided by the capabilities. See Annexe 1 for an explanation of the contents of all the plans.

Interface

The graphical interface, used to communcate with the user, has six different frames (for more details see 7.1) : *IntroW, CriteriasW, ResultW, TheoryBaseW, DefaultW, WaitW. CoordinW* co-ordinates these frames.

Events

The external events have already been identified previously. They are:

- *AnalyseCriteriasE* is generated each time there are changes in the preferences of the trip's criteria of the traveller. These events are generated by the GUI and handled by an agent's plan that is not part of a capability.
- *ChgeTicketE* is sent by a CompanyAgent when there are changes in a ticket's characteristics. This event is handled by the communication capability.
- *CancellationE* indicates that some travellers have cancelled their reservation for tickets for which the agent queues. This event is also handled by the communication capability.

The agent will co-ordinate all the capabilities so that each sub-goal will be correctly achieved. Some internal events will occur to announce the end of a capability's task. Figure 14 shows the relation between the events and the capabilities.
Finally, we can distinguish between ten internal events that the agent handles :

- *NewCriteriasE* occurs if the traveller changes the trip's criteria.
- *ChgeCriteriasE* occurs if the traveller changes his/her preferences.
- *ChgeGenCriteriasE* occurs if it is the first time the traveller asks for this trip.
- *TreatedE* occurs if a similar research has already been made.
- *TreatTicketE* states that the agent possesses all the information about the tickets available. The treatment of these data can begin..
- *DefaultE* states that the agent possesses information about each available ticket and that characteristics about the tickets have to be completed by the DefaultCap capability.
- *InitRulesE* states that all missing information concerning the ticket has been completed and that the Br capability can begin to search the "best ticket".
- *BookingE* states that the "best ticket" is known by the agent and that the communication capability can ask the CompanyAgent if there are vacant seats.
- *ComReplyVacancyE* states that the agent knows if there are available seats for the best tickets.
- *RemoveTickets* occurs if the agent has to start a search all over again because there are no available seats for the "best ticket".

The TravelAgent sends also external events to CompanyAgents. They are:

- *AskTicketE* announces that the agent asks the CompanyAgents if they possess tickets corresponding to given criteria.
- *AskVacancyE* announces that the agent asks the CompanyAgent if there are vacant seats for a ticket.
- *BookE* announces a CompanyAgent that the TravelAgent wants to book a ticket.
- *QueueE* announces a CompanyAgent that the TravelAgent wishes to queue for a ticket.
- *CancelE* announces that the TravelAgent wants to cancel a reservation.

These are the most important events but there are, of course, other events that occur in order to co-ordinate the plan within the capabilities. For a better comprehension, we can see how the most important events are issued on Figure 14.

Databases

Databases will correspond to each kind of beliefs described previously.
There are six kinds of beliefs and, as a consequence, the agent has at least six databases.

- *dbGenCriterias* stores the trip's criteria (departure, destination).
- *dbCriterias* stores the preferences of the traveller (smoker seat, business class, ...).
- *dbSociale* stores the name of the CompanyAgents (i.e. the airlines) of our system.
- *dbTickets* stores the information about the tickets received from the CompanyAgents.
- *dbBooked* stores the ticket that the TravelAgent books.
- *dbQueued* stores the tickets that the TravelAgent queues.

Of course, the TravelAgent will also use other databases to store the results calculated by the capabilities such as dbRules that stores all the formulas representing the characteristics of the tickets. These other data do not store new beliefs but store the same beliefs in other forms. They are:

- *dbRules* stores the formulas representing the characteristics of the tickets.
- *dbCriteres2* stores the criteria ordered according to the preferences of the user (after transformation -due to the problem of quantification- of the data contained in dbCriterias).
- *TicketBest* stores the best ticket that has been singled out by the search .
- *dbHistoric* stores the requests (general criteria + preferences criteria) that have already been done.
- *dbHistoricResult* stores the result of the searches corresponding to the requests stored in *dbHistoric.*
- *dbTickets2* stores the tickets and their characteristics (after the transformation –from *dbTickets*- due to the problem of quantification).

General schema

Annexe 2 presents 4 schemas describing more in detail the architecture of our agent system :

- Interconnection between capabilities, plans, events and databases : general view
- Interconnection between capabilities, plans, events and databases : the posting of the events
- Interconnection between capabilities, plans, events and databases : the treatment of the events
- Interconnection between capabilities, plans, events and databases : the access to the databases

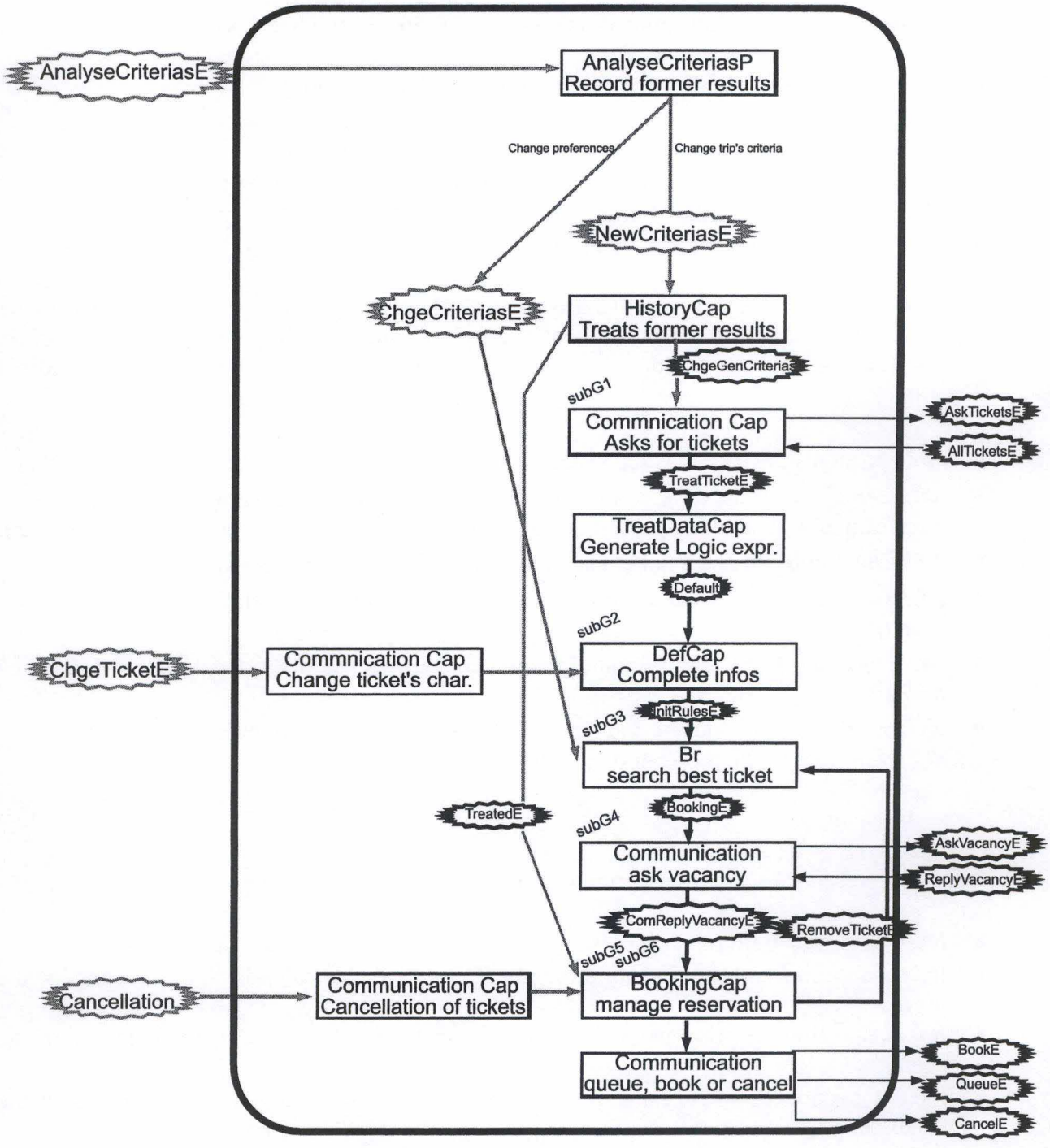


Figure 14 : How the capabilities are co-ordinated to achieve the sub-goals

6.3 CompanyAgent

6.3.1 Features of the CompanyAgent

The CompanyAgent is less complex than the TravelAgent, and hence its architecture is simple. To build the CompanyAgent, we follow the same procedure as for the TravelAgent : we have to identify the kinds of reasoning methods that will be used by the agent, the external events that drive the agent, the goals the agent can set for itself, the beliefs the agent has and finally, the functionalities that have to be provided by the agent.

Reasoning method

As we have already said, the CompanyAgent is relatively simple in our application. Therefore, the agent doesn't need to use complex reasoning methods. The CompanyAgent doesn't use non-monotonic logic techniques and, as a result, doesn't use Hades and Saten.

External events

As Figure 15 shows, the external events may come from the TravelAgent or from the company administrator. The TravelAgent may ask for information about tickets, make a cancellation, book a ticket or queue and ask if there are vacant seats. The administrator may change characteristics of tickets and, of course, has access to all the information that the CompanyAgent possesses.

The goals

The main goal of a CompanyAgent is to provide services so that the TravelAgent can possess all the information to choose a ticket and so that it can book the ticket. The sub-goals are "giving ticket's characteristics" (sub-goal1) and "managing reservations" (sub-goal2). The agent has also a sub-goal that updates beliefs about tickets when the company administrator changes some ticket features and that warns the TravelAgents of any change in a ticket's characteristics to TravelAgents (sub-goal3).

The beliefs

The beliefs of a CompanyAgent concern all the data that an agent has to know to achieve each sub-goal. We will take a closer look at each sub-goal now.

To give a ticket's characteristics (sub-goal1), the agent has to know the ticket's characteristics and the name of the TravelAgent that asks the information. To give information on vacant seats, the CompanyAgent has to know the number of reservations and the number of available seats for each kind of ticket. To manage the reservation (sub-goal 2), the agent has to know the name of the TravelAgents that had booked or queued for a ticket. To announce any change in a ticket's characteristics (sub-goal3), the CompanyAgent has to know which TravelAgent had asked information for this kind of ticket and it has to know what are the changes in the ticket's characteristics.

Functionalities

There are a few functions that the agent has to possess to achieve the sub-goals. Consequently, we will not regroup functions in the same way as we did with the TravelAgent. We can distinguish between some functions that are needed to achieve the sub-goals :

For sub-goal1 (i.e. give ticket's characteristics) :
- send general information to a TravelAgent
- send information about vacant seats

For sub-goal2 (i.e. manage reservation) :
- manage booked tickets
- manage queued tickets
- manage the cancellation of a ticket
- tell a TravelAgent that another TravelAgent has cancelled a reservation

For sub-goal3 (i.e. manage updating)
- remove a ticket from the beliefs and send the changes of a ticket's characteristics to the TravelAgents
- update the characteristics of a ticket and send the changes of a ticket's characteristics to the TravelAgents

6.3.2 Jack architecture

Now that we know what are the main features of the CompanyAgent, we can begin distinguishing between the different components of a Jack agent. This agent is very simple and there is no need to group a plan into capabilities. In other words, there is no capability in our CompanyAgent. We go on by defining what are the plans, the events and the databases.

Plans

A plan will correspond to each function described above. The functions for sub-goal2 are not implemented. Four plans will be implemented :

- *AskTicketP* treats a demand of information about a ticket.
- *AskVacancy* treats a demand of information about vacant seats.
- *UpdateP* updates the databases if the administrator has made some changes and sends the changes to some TravelAgents.
- *RemoveP* removes some tickets on the administrator's demand and sends the changes to some TravelAgents.

See Annexe 1 for an explanation of these plans.

Events

The external events have already been identified previously. A CompanyAgent handles four external events. They are:

- *AskTicketE* announces that a TravelAgent asks tickets corresponding to general criteria.

- *AskVacancyE* announces that a TravelAgent asks if there are vacant seats for a ticket.
- *UpdateE* announces that the administrator has made some changes in the tickets' characteristics.
- *RemoveE* announces that the administrator has removed a ticket from the proposition.
- *BookE* announces that a TravelAgent wishes to book a ticket.
- *QueueE* announces that a TravelAgent wants to queue for a ticket.
- *CancelE* announces that a TravelAgent cancels its reservation for a ticket .

The CompanyAgent sends some events to the TravelAgent. They are:

- *AllTicketsE* tells the TravelAgent that the CompanyAgent has sent all its tickets (corresponding to the general criteria specified by the user).
- *ReplyVacancy* replies a CompanyAgent whether or not there are vacant seats for a ticket.
- *ChgeTicketE* is sent by a CompanyAgent when there are changes in ticket characteristics. This event is handled by the communication capability.
- *CancellationE* indicates that some travellers have cancelled their reservation for tickets.

There are no relevant internal events in a CompanyAgent because the structure of the agent is too simple.

Databases

The database corresponds to the agent's beliefs.

- *dbTicketsComp* contains all the information about a ticket (the departure, arrival, vacancy seats, duration of the trip, etc.).
- *dbCalls* contains the names of TravelAgents that asked for information and the content of the request (i.e. departure and arrival of the trip).
- *dbbook* contains all the reservations made for the tickets of the company.
- *dbqueue* contains all the names of the TravelAgents that are waiting for some kind of ticket.

Interface

A graphical interface will be used to communicate with the company administrator. Only one frame composes the GUI. It is called *CompanyAgentW*.

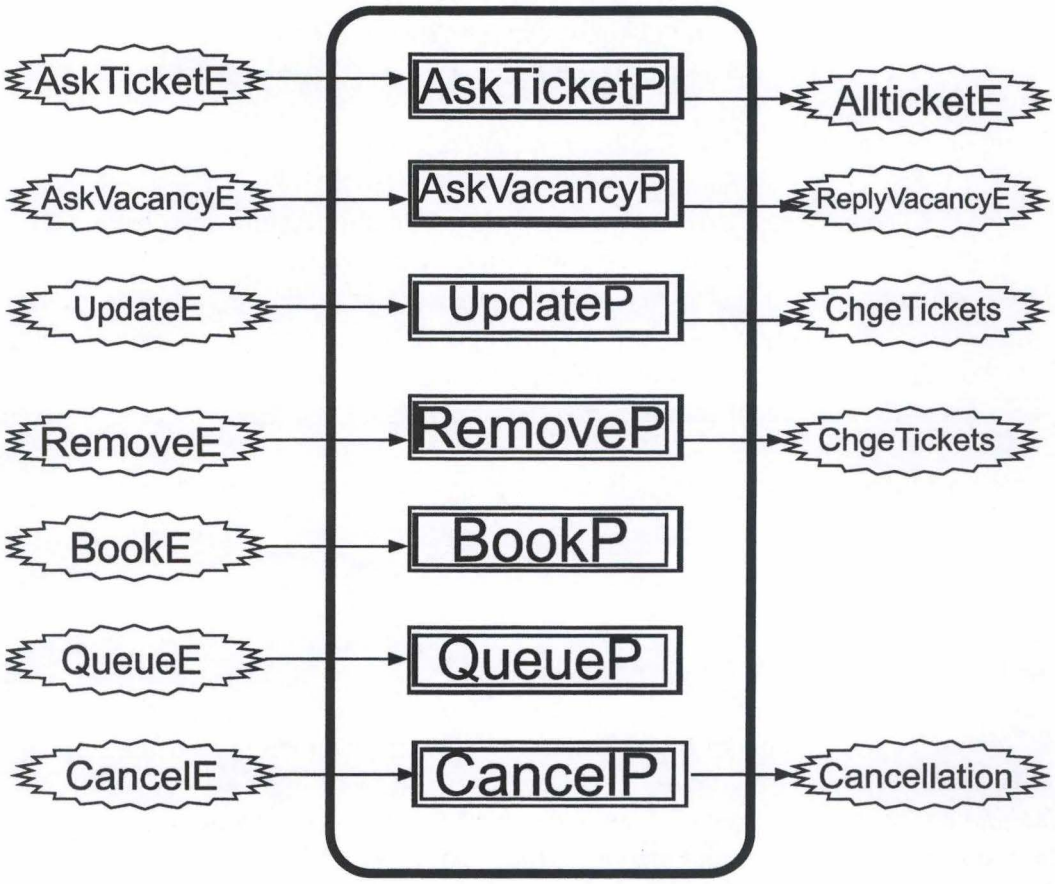


Figure 15 : The plans and the events of a CompanyAgent

7. Implementation

We discuss in this chapter the implementation of our agents. We begin by describing the interface created for each agent. Second, we explain how we use non-monotonic logic and finally, we present some examples of Jack components of our agents.

7.1 Graphical User Interface

In this section, we will first talk about the Graphical User Interfaces. This is an important aspect of an agent since it constitutes the visible part of it. The traveller has, in a way, a privileged link with his/her personal agent and hence the attention given to the GUI can be decisive.

The design rules used for other application interfaces have always to be respected. It is not because the programming approach is different that we can forget the design rigor. Nevertheless, particular attention has to be paid to an agent's interface in order to respect the agent's properties such as knowing the individual it is assisting.

Designing a good user interface was not our main preoccupation. It was, however, necessary to reflect correctly our agent's capabilities. While designing the GUI of the TravelAgent, we encountered two kinds of problems. The first one is the fact that the traveller needs to understand correctly what the agent can do. The second problem concerns the fact that the traveller has to be able to express easily his or her preferences. To design the GUI of the CompanyAgent, our only preoccupation was to be able to view and to change easily the data contained in the agent.

Since Jack and Java are compatible, the GUIs are written in Java.

7.1.1 The TravelAgent interface

The TravelAgent GUI is composed of four frames. The first one is just a frame with a button "Tickets search" which is used to initialise the agent.

The second frame gives the traveller the opportunity to fill in the criteria of the trip (i.e. departure, destination and date) and his/her personal preferences. Is the price of the ticket the only criterion for his/her choice? Will he or she accept to pay more if the flight takes less time? Does he/she mind if he/she can't smoke in the airplane? Below, all the attributes that are used as a characteristic of a ticket and as criterion of a traveller are listed along with their meaning.

Attributes used as a characteristic of a ticket :

- **Price (integer)** : the price of the ticket
- **Length (double)** : the length (in time) of the flight
- **Smoker (Boolean)** : is positive if the ticket can correspond to a smoker seat in the airplane
- **Champagne (Boolean)** : is positive if the ticket corresponds to a flight during which the traveller will be served champagne

- **Business (Boolean)**: is positive if the ticket is a business class ticket
- **TV (Boolean)** : is positive if the ticket corresponds to a flight in an airplane in which there are personal televisions

Attributes used as criterion :

- **Price (double)** : the degree of importance attached to the price of the ticket
- **Length (double)** : the degree of importance attached to the length of the flight
- **Smoker (double)** : the degree of importance attached to the fact that the ticket corresponds to a smoker seat
- **Champagne (double)** : the degree of importance attached to the fact that champagne is served during the flight corresponding to the ticket
- **Business (double)** : the degree of importance attached to the fact that the ticket is a business class ticket
- **TV (double)** : the degree of importance attached to the fact that there are personal televisions in the airplane corresponding to the ticket

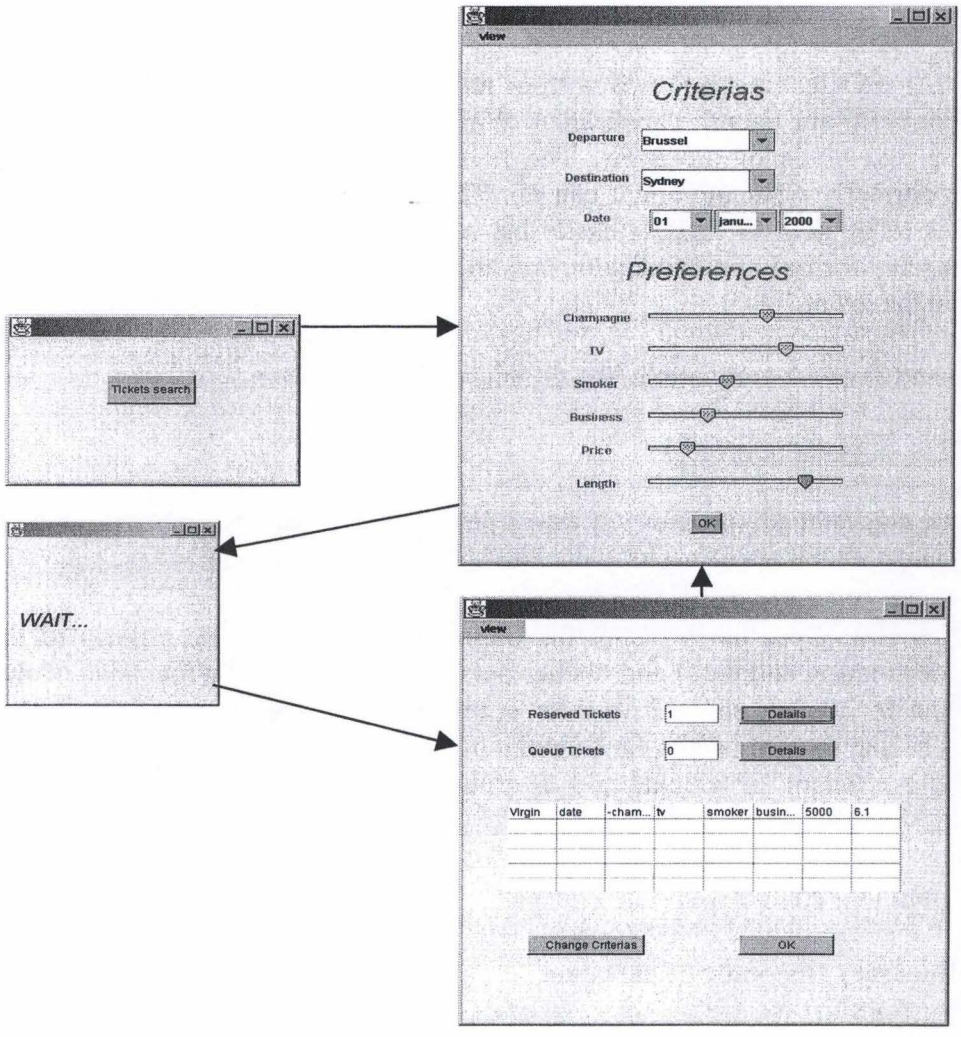


Figure 16 : The series of the TravelAgent's GUIs

When the traveller pushes on the "OK" button, the task of the TravelAgent is to find the ticket that matches the best the preferences expressed by the traveller. When the best ticket is found, the agent looks for a vacancy for the chosen flight. If there is a seat vacant, the agent books it, otherwise it queues for that ticket and looks for another best ticket.

When the agent has one ticket booked, it gives the results with the last frame. The traveller can view the booked and queued tickets with all their features. From this frame, the traveller can have access to the criteria frame. So, he can restart a new search.

7.1.2 The CompanyAgent interface

There exists only one frame for the CompanyAgent. This frame allows the company to perform four operations (view features of a ticket, update the ticket's characteristics, remove a ticket, add a new ticket). Each operation can be performed using the tree buttons contained in the frame.

To view the features of a ticket, the ticket has to be selected in the list and the "Get" button has to be clicked upon. The features will appear on the left side of the frame. If the traveller wants to change the characteristics of a ticket, he has to view first the ticket's feature. After modifying some fields on the left side of the frame, clicking on the "Update" button, will record the change. The operation can also consist in removing a ticket, the traveller then has to select the ticket in the list and press the "Remove" button.

In order to add a ticket, one has to fill the empty text fields before pushing the "Update" button.

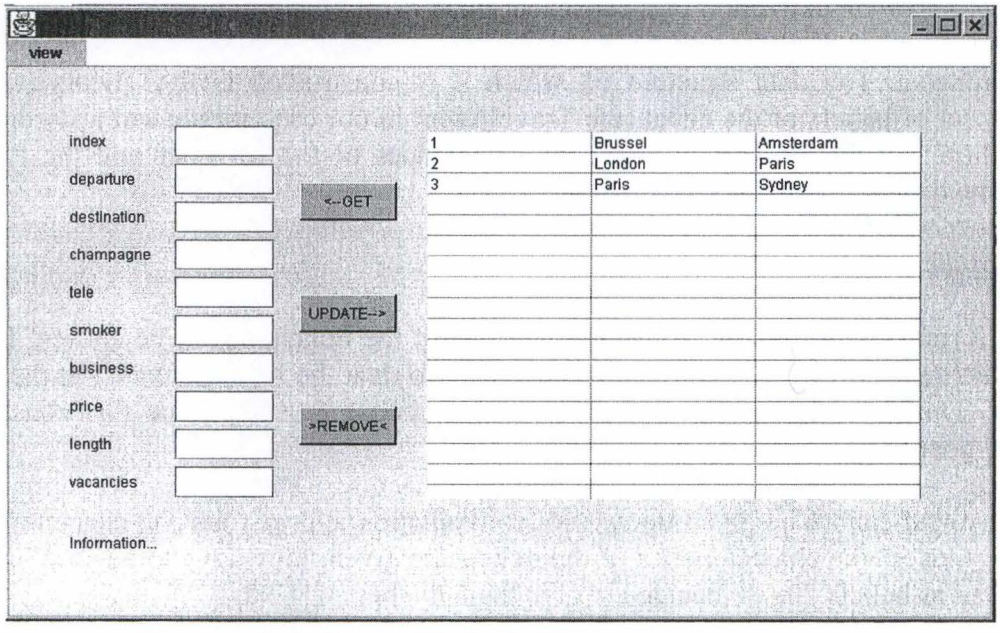


Figure 17 : The CompanyAgent's GUI

7.2 Non-monotonic reasoning in our agents

7.2.1 How do we use Belief Revision?

We use Belief Revision in two different ways. The first way is a degenerated use of Belief Revision. Normally, one has a belief set and revising these beliefs means adding a new belief and then extracting a consistent theory (in such a way that sets how to treat the inconsistencies) from the enlarged belief set. We make use of the Belief Revision principles, not to really *revise* a belief set because we do not always maintain a consistent belief set. In our application, we fill a data structure that we consider as our belief set, and only when it contains all the elements desired (characteristics of the tickets, preferences of the user, etc.), we extract a consistent theory from it. Our method for the selection of the best ticket rests on this kind of use of Belief Revision. All the knowledge about the tickets and the preferences form the belief set and the selection take the form of an extraction of a consistent theory from this belief set.

We also use Belief Revision in a more classical way. This time the belief set is really *revised* with changes in the beliefs. Indeed, imagine the case of a traveller who uses our Intelligent Agent to buy the flight ticket that matches the best his preferences. He or she changes his or her mind concerning his/her preferences when the ticket is already found. Using Belief Revision avoids that the Intelligent Agent spawns the whole treatment from scratch. Only the elements of its belief set concerned would be changed. In this case, the extraction of a consistent theory can be executed to single out one ticket.

We present here how the extraction of a consistent theory from the belief set of the agent can lead to the selection of the best ticket. The tool we used to manage the belief set and the extractions is Saten. The data structure on which it is constructed is the TheoryBase. It represents the set of beliefs of the agent (the TravelAgent in our case) organised according to their rank. These are mainly beliefs about the preferences of the traveller and the tickets proposed by the different airlines.

The ranking of the beliefs

First, an OR-formula expresses the constraint that one of the tickets has to be chosen. It has the form *'ticket1|ticket2|ticket3...'* and will be necessary to treat the inconsistencies at the time of extraction. This formula has the highest rank (we choose 0.9999) since this constraint represents the goal of the search.

Something else that cannot be given up when a contradiction appears, are the characteristics of the tickets. Indeed, the characteristics of the tickets are given, they have to be seen as true facts rather than as beliefs. So we decided to give them the rank 0.9998.

Next come the criteria, ranked between 0.9 and 0.1. Their rank depends on the preferences specified by the traveller. For example, if the traveller classifies his or her preferences in such a way that 'champagne' is assigned a rank 0.9 and 'smoker' a rank 0.4, the TravelAgent must be convinced that the traveller attaches more importance to the fact that champagne will be served on board of the plane than that he/she will be given a smoker seat.

Finally, we decided to put the name of the tickets at the bottom of the TheoryBase. They are assigned the lowest rank (0.0001). They are the first thing that the agent will get rid off when there will be an inconsistency. It can be interpreted as follows : before starting the extraction of a consistent theory from the TheoryBase, the TravelAgent beliefs very weakly that the tickets will be chosen. It knows for sure that one of them will be singled out (the OR-formula is assigned the rank 0.9999) but it does not know which one. So it admits the possibility that 'ticket1' will be chosen as well as the possibility that 'ticket2' or 'ticket3' will be chosen. It does not belief in these formulas strongly, it just admits the possibility of their veracity.

In order for the whole thing to work, we had to put the characteristics of the tickets under the form of rules of the following type "*ticket->attribute*". Indeed, this represents the only way inconsistencies can appear and the best ticket can be singled out. For example, when using such a system, we can have :

```
(1) ticket1->champagne          one characteristic of ticket1
(2) ticket2->-champagne         one characteristic of ticket2
(3) champagne                one traveller's preference
(4) ticket1                     one candidate ticket
(5) ticket2                     one candidate ticket
```

If these five formulas represent a belief set, two kinds of inconsistencies can be found. First, (2) contradicts (3) if (5) is true (i.e. it is kept in the belief set). Indeed, "ticket2->-champagne" coupled with "ticket2" would imply "-champagne" which is exactly the inverse of "champagne".

The second type of inconsistency can appear if we use the contra-position of the formulas representing the characteristics of the tickets. In our example, the contra-position of (1) and (2) would be :

```
(1b) -champagne -> -ticket1
(2b) champagne -> -ticket2
```

If we confront them with "champagne" (3), there is a contradiction between (2b) and (5). More explanations concerning the way it works will be given later on.

Below, we list simplified example of our TheoryBase (before extraction) in the case of 2 tickets proposed.

Example of our TheoryBase (before extraction) in the case of 2 tickets

ticket1|ticket2	0.9999
ticket1->champagne	0.9998
ticket1->goodprice	0.9998
ticket1->-smoker	0.9998
ticket2->-champagne	0.9998
ticket2->goodprice	0.9998
ticket2->smoker	0.9998
goodprice	0.9
champagne	0.7
smoker	0.4
ticket1	0.0001
ticket2	0.0001

How is executed the extraction from this TheoryBase? The first question that has to be asked is "Are there any inconsistencies?". The answer is "yes". If we use the contraposition of the formulas ranked at 0.9998, we obtain :

```
-champagne ->-ticket1
-goodprice -> -ticket1
smoker -> -ticket1
champagne -> -ticket2
-goodprice -> -ticket2
-smoker -> -ticket2
```

Coupled with the formulas representing the criteria, we have :

```
smoker
smoker->-ticket1

champagne
champagne->-ticket2
```

It appears that both ticket1 and ticket2 have to be negated. This is impossible, however, since the highest ranked formula ticket1|ticket2 that constrains one of the 2 tickets has to be true. So, in order to get rid off the inconsistency, one or more formulas have to be given up: either the OR-formula, or one formula representing the tickets' characteristics, or one of the criteria, or one ticket. Of course, we would like to abandon one of the tickets, which has the lowest rank. But it is not sufficient because we do not know which one to give up. As a consequence, we have to climb up in the hierarchy and we have to take a look at the criteria. By so doing,

we see that 'smoker' is ranked lower than 'champagne' and thus we will give up 'smoker'. Clearly, this means that 'ticket1' which is 'champagne' but not 'smoker' is preferred to 'ticket2' which is 'smoker' but not 'champagne'. However, dropping 'smoker' is not enough to get rid off the inconsistency because the following formulas still constrain one ticket to be dropped.

```
-champagne -> -ticket1
-goodprice -> -ticket1
-goodprice -> -ticket2
-smoker -> -ticket2
```

As 'champagne' has not been abandoned, 'champagne->-ticket2' obliges us to choose 'ticket2' as the ticket to give up.

Hence, after extraction, our TheoryBase looks like :

```
ticket1|ticket2          0.9999
-------------------------------
ticket1->champagne       0.9998
ticket1->goodprice       0.9998
ticket1->-smoker         0.9998
ticket2->-champagne      0.9998
ticket2->goodprice       0.9998
ticket2->smoker          0.9998
-------------------------------
goodprice                   0.9
champagne                   0.7
-------------------------------
ticket2                  0.0001
```

Apart from the OR-formula and the characteristics of the tickets, what remains is the best ticket (ticket1) and the criteria that it fulfils ('goodprice' and 'champagne').

The problem of optimisation

In the previous example, it was obvious that the best ticket was 'ticket1' because the criterion that it did not possess ('smoker') was less important (i.e. lower ranked). But it is not always that simple.

Another example of our TheoryBase

ticket1|ticket2	0.9999
ticket1->-champagne	0.9998
ticket1->goodprice	0.9998
ticket1->-smoker	0.9998
ticket2->champagne	0.9998
ticket2->-goodprice	0.9998
ticket2->smoker	0.9998
goodprice	0.9
champagne	0.7
smoker	0.4
ticket1	0.0001
ticket2	0.0001

If we consider the TheoryBase above, we see that 'ticket1' possesses the most important criterion for the traveller ('goodprice') but that it does not have the two other criteria. 'ticket2' does not have the characteristic 'goodprice' but it does possess the two others. Which ticket do we have to choose?

We have the opinion that, 'ticket2' should be chosen. Indeed, the goal of the TravelAgent should be to optimise the traveller's satisfaction, according to the preferences he/she expresses. In this case, we can say that fulfilling the criterion 'goodprice' has a weight of 0.9, while fulfilling the criteria 'champagne' and 'smoker' has a weight of 0.7 + 0.4 = 1.1! Clearly, it embodies the principle of Minimal Change that should always be satisfied in Belief Revision, at least if we consider the minimality in terms of cumulated ranks. The problem is that we did not have the adequate Extractor at our disposition. Saten doesn't possess an Extractor that is able to realise such an optimisation. The more suitable Extractor that Saten proposes is Maxi-Adjustment. In the example discussed, Maxi-Adjustment chooses 'ticket1' because it satisfies the highest ranked criterion. If ever the two tickets satisfy the highest ranked criterion the same way (i.e. they both fulfil it or they both do not fulfil it), then the way they satisfy the second highest ranked criterion is analysed. If the two tickets are still tied, the third highest ranked criterion is analysed, and so on.

The problem of perfect equality

Now, we will look at an example where two tickets seem to satisfy the traveller equally. Let's consider the TheoryBase listed below.

Example of our TheoryBase

ticket1|ticket2	0.9999
ticket1->-champagne	0.9998
ticket1->goodprice	0.9998
ticket1->smoker	0.9998
ticket2->champagne	0.9998
ticket2->goodprice	0.9998
ticket2->-smoker	0.9998
goodprice	0.9
champagne	0.4
smoker	0.4
ticket1	0.0001
ticket2	0.0001

Both 'ticket1' and 'ticket2' fulfil the most important criterion ('goodprice') and both fulfil another one, respectively 'smoker' and 'champagne', ranked at the same degree! So, which one has to be kept and which one has to be given up? The more logical answer would be 'none'. That's why we used the Maxi Adjustment preferably to Standard Adjustment. In this kind of situation, Maxi Adjustment keeps both tickets while Standard Adjustment removes both of them. In fact, the two tickets have to be kept but one of them will be chosen at random to be booked. If there is no vacancy for that ticket, another search (without that ticket) will be spawned and the ticket found will be the other one.

The problem of the quantifications

A problem appears when we have to deal with quantitative attributes of the tickets (i.e. the price of the ticket and the length of the flight in our application) even when our TheoryBase works only with Boolean formulas. Up to now, in our examples, we used extremely simplified formulas like 'goodprice'. But we cannot stop there. The main difficulty is to establish a correspondence between the quantitative criteria and the quantitative attributes of the tickets. If we simply write the quantification under the form of a Boolean expressing the exact quantification (e.g. 'priceIs5000'), the correspondence is impossible to make because there should then exist a huge list of criteria representing all the possible prices (e.g. 'priceIs5000', 'priceIs5001', 'priceIs5002', ...). Practically, this is too difficult to implement.

The solution to this problem is using intervals. The price to pay is the loss of accuracy. In the list of the tickets' characteristics, instead of a precise formula like 'ticket1->priceIs5000', there

will be a formula of the type 'ticket1->price2' expressing that the considered price is situated in the second interval, according to the prices of all the proposed tickets.

But how are the intervals calculated The problem of the price of the tickets has to be examined. We consider the higher bound as the price of the cheapest ticket and the lower bound as the price of the most expensive ticket. The interval between these two bounds is divided by 5 to obtain 5 intervals of the same size. If one wants to gain accuracy, one can use more than 5 intervals but then the time of computation will suffer from it. A compromise has to be found. Let's have a look at an example with 3 tickets.

Example

```
ticket1 costs 5000F
ticket2 costs 5410F
ticket3 costs 7000F

The lower bound is 5000F
The higher bound is 7000F

The intervals are :
     price1 : from 5000F to 5400F
     price2 : from 5401F to 5800F
     price3 : from 5801F to 6200F
     price4 : from 6201F to 6600F
     price5 : from 6601F to 7000F
And the formulas are :
     ticket1->price1
     ticket2->price2
     ticket3->price5
```

Obviously, using such intervals is quite arbitrary. Two tickets that differ slightly, say several francs, can be classified in 2 different intervals, what could incite the agent to choose cheaper one if the price is the first criterion for the traveller, even if the more expensive one has other characteristics that are of interest for the traveller. What happens if in the previous example a new ticket, 'ticket4' costs 5395F, is added? Even if this ticket is only 15F cheaper than 'ticket2', it fallss in another interval ('price3').

In order for the whole TheoryBase to work, a subtlety has to be found. When the traveller gives a degree of preference to a quantifiable criterion, it doesn't give birth to one formula but to 5 formulas. For example, if the criterion 'price' is assigned the degree 0.9, this will create the following formulas :

```
price1 of degree 1.0 * 0.9 = 0.9
price2 of degree 0.8 * 0.9 = 0.72
price3 of degree 0.6 * 0.9 = 0.54
price1 of degree 0.4 * 0.9 = 0.32
price1 of degree 0.2 * 0.9 = 0.18
```

In parallel, something has to be done with the formulas representing the characteristics of the tickets. For a given ticket, there must be as many formulas as there are categories. A positive formula and 4 negative ones. Coming back to our example, the 'ticket1' 's characteristics formulas will be :

```
ticket1 -> price1
ticket1 -> -price2
ticket1 -> -price3
ticket1 -> -price4
ticket1 -> -price5
```

Let's now illustrate this with an example of two tickets and 3 intervals of prices. For more clarity, we will limit ourselves to the formulas concerning the price of the tickets.

```
ticket1 costs 5000F and will be classified in the first interval
(price1)
ticket2 costs 6000F and will be classified in the first interval
(price3)

This will give the following TheoryBase:

                    ticket1|ticket2 0.9999
                    ---------------------

                    ticket1->price1 0.9998
                    ticket1->-price20.9998
                    ticket1->-price30.9998
                    ticket2->-price10.9998
                    ticket2->-price20.9998
                    ticket2->price3 0.9998
                    ---------------------

                    price1           0.9
                    price2          0.72
                    price3          0.54
                    ---------------------

                    ticket1        0.0001
                    ticket2        0.0001
```

In order to extract a consistent theory from this TheoryBase, some formulas have to be given up. Again, we make use of the contra-position of the tickets' characteristics formulas.

```
-price1 -> -ticket1
price2  -> -ticket1
price3  -> -ticket1
price1  -> -ticket2
price2  -> -ticket2
-price3 -> -ticket2
```

First, whatever the ticket that will be chosen, the formula 'price2' must be given up. The reason lies in the two contraposition formulas 'price2->-ticket1' and 'price2->-ticket2' that constrain the two tickets to be given up, what is inconsistent with the OR-formula. Afterwards, 'price3' and 'ticket2' will be given up to 'price2' and 'ticket1' because 'price3' is lower ranked than 'price2'.

For an example of a complete and more realistic TheoryBase (before and after extraction), see Annexe 4.

The analysis of the TheoryBase after extraction

Once the extraction is made, the agent has to analyse the TheoryBase to see what the remaining ticket (i.e. the best ticket) is and what its characteristics are.

First, it has to retrieve the number of the "best ticket" which is normally in the last position in the TheoryBase. Then, it has to consider the characteristics of that ticket. But all the information about the ticket is not explicitly in the TheoryBase. The negative attributes must be deducted from the absence of the corresponding positive attributes in the TheoryBase. For example, if "champagne" has disappeared from the TheoryBase, it means that the chosen ticket has the attribute "-champagne".

An analysis of the TheoryBase (after extraction) represented in Annexe 4 follows :

```
THE BEST TICKET IS ticket1
HERE ARE THE DETAILS OF THIS TICKET  :
     price1
     business
     length5
FILLING OF THE NEGATIVE FIELDS.
      CHAMP : -champagne
      TV : -tv
      SMO : -smoker
      BUSI : business
      PRICE : price1
      LENGTH : length5
```

7.2.2 How do we use Default Logic?

Our TravelAgent is able to treat Default Logic thanks to Hades. The main data structure of that tool is the DefaultTheory. It is composed of an array of *strings* (*in*) for the beliefs (or facts), an array of *default* objects (*defaults*) for the defaults and an array of arrays of *String* (*extensions*) for the extensions.

The facts are composed of information about the traveller's preferences and the tickets. Remember that, for the purpose of the treatment by the Belief Revision module, the information about the tickets are under the form of a rule of the type "*ticket->arrtibute*".

Here are the default rules we used. They can be of two types. The first type of rules are those with a negative consequent.

For example, for i ranging from 1 to the number of tickets proposed :

```
"","-(ticket"+i+"->business)&-(ticket"+i+"->champagne)"
-------------------------------------------------------
                "ticket"+i+"->-champagne"
```

which is read : If it is consistent to assume "-(ticket"+i+"->business)&-(ticket"+i+"->champagne)", then conclude "ticket"+i+"->-champagne".
More intuitively, it means that normally, in non business class, no champagne is served on board.

The other default rules of this type are :

```
"","-(ticket"+i+"->business)&-(ticket"+i+"->tv)"
------------------------------------------------
                "ticket"+i+"->-tv"

"","-(ticket"+i+"->business)&-(ticket"+i+"->smoker)"
----------------------------------------------------
                "ticket"+i+"->-smoker"
```

The second type of rules are those with a positive consequent. For example,

```
"(ticket"+i+"->business)","-(ticket"+i+"->-champagne)"
------------------------------------------------------
                "ticket"+i+"->champagne"
```

which is read : If "(ticket"+i+"->business)" and if it is consistent to assume "-(ticket"+i+"->-champagne)" , then conclude "ticket"+i+"->-champagne".
More intuitively, it means that normally, in business or unknown class, champagne is served on board.

The other default rules of this type are :

$$\frac{\text{"(ticket"+i+"->business)","-(ticket"+i+"->-tv)"}}{\text{"ticket"+i+"->tv"}}$$

$$\frac{\text{"(ticket"+i+"->business)","-(ticket"+i+"->-smoker)"}}{\text{"ticket"+i+"->smoker"}}$$

See Annexe 5 for an example of default theory.

7.3 Example of Jack components

In this section, we take a closer look at the code of our agents and we show how to build the Jack objects forming the agents. In chapter 4, we have described the different entities of a Jack agent. We present in this section an example of code for each type of entity. First, we present four components of the CompanyAgent : the **Agent** itself, an **Event** (*AskVacancyE*), the corresponding **Plan** (*AskVacancyP*) and a **Database** (*TicketsCompDB*). Finally, we describe a **Capability** of the TravelAgent (*DefCap*). For more explanation of the code of all the plans, see Annexe 1.

7.3.1 An agent : CompanyAgent

The code of the "*CompanyAgent*" declares which databases the agent uses, which events it sends or receives and which plans it can execute. The code contains also the methods of the agent.

Declaration of the agent "CompanyAgent"

The first thing to do when building an entity "*Agent*" is declaring it. *"Agent"* is a JAL keyword used to introduce an agent definition. "*CompanyAgent*" is the name of our agent. "*extends Agent*" plays the same role as in Java; it indicates that the agent being defined inherits from a JAL base class called "*Agent*". "*implements AGENTComp*" states that that agent implements a given Java interface.

```
public agent CompanyAgent extends Agent implements AGENTComp {
```

Jack statements (database, events, plans)

Next, we have to specify the **databases** the agent uses. These statements describe to which database the agent has access. Each database definition provides a database relation that the agent can use to express and store information. "*TicketCompDB*" defines the type of the database relation, whereas "*dbMyTickets*" is the name used to identify the instance of the relation. The adjective "private" means that the agent has private access to the database relation "*dbMyTickets*". This means that the agent has its own copy of the relation, which it can read and modify independently of all the other agents, even those of the same agent class.

A database relation can be declared "*private*", "*agent*" (shared with all the agents of the agent class) or "*global*" (shared with all the agents of all the agent classes).

```
#private database TicketsCompDB dbMyTickets();
#private database DBCalls dbCalls();
```

The declaration of the **events** the agent handles or posts are as follows:

"*#handles event*" statements identify the events which the agent will attempt to respond if they arise. By handling an event, the agent claims to have at least one plan available that it can execute when this event arises.

```
#handles event AskTicketE;
#handles event AskVacancyE;
```

"*#posts event*" statements describe the event that the agent can post internally (to be handled by other plans). For example, "*InitDataCompanyE*" is the name of the event and "*data*" is a local variable that will be used to reference that event when it will be posted.

```
#posts event InitDataCompanyE data;
#posts event GetE getE;
#posts event UpdateE updateE;
#posts event RemoveE removeE;
#posts event FillTableE fill;
```

The "*#uses*" statements conclude the JAL declarations. The "*#uses plan*" statements identify the **plans** that the agent can execute to handle all the events that we just declared. The string that follows "*plan*" in the statement is the name of the plan. This means that all the instances of the agents have access to the specified plan.

```
#uses plan InitDataCompanyP;
#uses plan AskTicketP;
#uses plan AskVacancyP;
#uses plan GetP;
#uses plan UpdateP;
#uses plan RemoveP;
#uses plan FillTableP;
```

Java statements

Now that the JAL statements have been written, we have to specify some **Java statements**. The two following lines of code declare two variables local to the agent.

```
private String name;
private CompanyInterface interfac;
```

Methods

After the declarations, the **methods** of the agent have to be written down.
The "*CompanyAgent*" method creates the agent:

```
public CompanyAgent(String s,CompanyInterface j)
                { ... }
```

The "*search*" method below is used to send a "*data*" event. "*postEventAndWait"* is similar to "*postEvent*", except that, instead of posting the event asynchronously, it is posted synchronously. The agent still sends the event in a separate task, but it has to wait until this event has been treated. "*data*" is the "reference name" that has been given to an occurrence of the event "*InitDataCompanyE*". "*fill*" is the "posting name" in the declaration of the event "*InitDataCompanyE*". Posting that event means calling the execution of the plan ("*InitDataCompanyP*" in this case) that handles it.

```
public void search()
        { postEventAndWait(data.fill()); }
```

"*getName*" is a method, used when one wants to access the name of the agent. This way of doing is better than directly accessing the variable "*name*". Indeed, it is always better to let the variables of the agent private and to allow their access via a "*public*" method.

```
public String getName()
        { return(name); }
```

The method "*getHardware*" enables the access to the interface of the agent.

```
public CompanyInterface getHardware()
        { return interfac; }
```

"*getTicket*" is a method called from the interface when one wants to display the characteristics of a selected ticket.

```
public void getTicket(int i)
        { postEventAndWait(getE.post(i));}
```

This method ("*updateTicket*") is also called from the interface, when ticket changes are specified. i, dep, dest, c, t, s, b, p, h and v are the arguments of the event. These arguments must have been declared in the definition of the event "*UpdateE*".

```
public void updateTicket(int i,String dep,String dest,String c,
                                String t,String s,String b,int p,double h,int v)
        { postEventAndWait(updateE.post(i,dep,dest,c,t,s,b,p,h,v)); }
```

The method "*removeTicket*" is used to remove a ticket from the company's database instead of updating it (as in the previous method).

```
public void removeTicket(int i)
        { postEventAndWait(removeE.post(i)); }
```

Finally, the method "*fiilT*" is called when one wants to fill the tables of the interface with the tickets contained in the company's database.

```
public void fillT()
        { postEventAndWait(fill.post()); }
}
```

7.3.2 An event : AskVacancyE

The event code specifies which arguments are sent with the events and the methods that allow to send the event.

Declaration of the event "AskVacancy"

The "*AskVacancyE*" event is a "*message*" event. A "*message*" event is an event that is sent externally by an agent to another agent, in this case, by a TravelAgent to a CompanyAgent. It embodies the question "Is there a vacancy for the ticket X". The **declaration of the event** "*AskVacancyE*" is the name and "*MessageEvent*" the base class it extends.

```
event AskVacancyE extends MessageEvent {
```

Declaration of variable

Some **declarations of variables** that compose the event are as follows:

```
public String dep,dest,ch,tv,sm,bu;
public int pri;
public double len;
```

Posting method

After this has been done, one has to declare the event's **posting method**. This means one has to describe how the event is constructed and then posted.

```
#posted as post(String depar,String destin,String champ,String telev,String smok,String
busin,int price,double length)
```

Body

The **body** of the event consists of the assignation of the values given in the parameters of the posting method to the variables of the event.

```
{ dep = depar;
  dest = destin;
  ch = champ;
  tv = telev;
  sm = smok;
  bu = busin;
  pri = price;
  len = length;}
```

7.3.3 A plan : AskVacancyP

The plan code specifies which events it sends or posts and which databases it accesses. The plan's body specifies the treatment executed in response to an event.

Declaration of the plan "AskVacancyP"

"*AskVacancyP*" is a plan of the CompanyAgent that corresponds to the event "*AskVacancyE*". It answers to a TravelAgent that asked whether there is vacancy for a certain ticket. "*AskVacancyP*" is the name and "*Plan*" the base class it extends.

```
plan AskVacancyP extends Plan {
```

Jack statements (events and database)

Most #-declarations are optional in a plan definition, the "*#handles event*" declaration, however, is mandatory. It specifies the **event** the plan handles, "*AskVacancyE*" in this case. Whenever an instance of this event occurs, the agent triggers "*AskVacancyP*".

```
#handles event AskVacancyE ask;
```

The "*#modifies database*" declaration identifies a database relation the plan modifies. It means that the plan will use the database relation's base methods "*assert()*" and "*retract()*" .

```
#modifies database TicketsCompDB dbMyTickets;
```

The following declaration indicates that the **plan** requires a particular Java interface ("*AGENTComp*") when it is executed.

```
#uses agent implementing AGENTComp agen;
```

The "*#sends event*" declaration identifies the **events** that the plan can post externally (i.e. to other agents). The event that will be sent must be of the type "*message event*".

```
#sends event ReplyVacancyE reply;
```

Body

Now that the JAL declarations are dealt with, we proceed with the real content of the plan. The "*body()*" method is the top-level reasoning method of Jack and is always executed whenever the plan is executed. It is just like the "*main()*" method in Java.

```
body()
{
```

The following declaration initialises the variable "*interf*" by assigning it the interface (in which the plan will write something) by invoking the method "*getHardware*" of the agent. The two others extract the name of the agent that sends the message event "*AskTicketE*" from its address. Indeed, the address of an agent is of the type "*LocalName@PortalName*" where "*LocalName*" is the name that identifies the agent at a given portal on the remote agent communications network (e.g. "Sabena"). "*PortalName*" is the name of the portal to which the agent listens and which it uses to communicate with other agents on the remote agent communication network. "*ind*" is assigned the index of the character "@" in the full address and "*agency*" is assigned the sub-string before that character (i.e. the "*LocalName*").

```
CompanyInterface interf = agen.getHardware();
int ind = ask.from.indexOf("@");
String agency = ask.from.substring(0,ind);
```

The line "*Logical int i, vacancy*" declares the variables *i* and *vacancy* as Logical int. Logical variables follow the semantic behaviour of variables from logic programming languages such as Prolog, that can be unified. An unbound logical variable is a variable whose value is still unknown to the agent, whereas a bound logical variable is a variable whose value has been determined.

In this case, the agent wants to find the value of the field vacancy of the database relation "*dbMyTickets*". It uses the "*get*" method that is defined in the database relation that tells the agent how the database will be accessed to unify these two variables with the value they have in the tuple. The tuple represents an element of the database relation and is characterised by the attributes of "*ask*", the message event containing the characteristics of the tickets for which the question of vacancy is asked.

```
Logical int i, vacancy;
dbMyTickets.get(i,ask.dep,ask.dest,ask.ch,ask.tv,ask.sm,ask.bu,ask.pri,ask.len,vacancy);
```

The instruction hereunder calls the method "*info*" of the interface "*interf*" and displays the message "send tickets to AgencyName".

```
interf.info("send tickets to " + agency);
```

The following lines represent the sending of the answer to the TravelAgent that issued the initial request. The two first lines deal with the conversion of the "*logical Boolean vacancy*" into a "*Boolean (vacant)*". "*@send*" is a JAL-statement that is used to send a message event to another agent from within a reasoning method (remember that "*body()*" is a reasoning method). Like the "*@post*" statement, it uses one of the message event's own posting methods ("*post*" in this case). Note that the reply consists not only of the Boolean but also of the characteristics of the tickets (contained in the received message event "*ask*") so that the TravelAgent that will receive the reply will know to which ticket the reply corresponds.

```
Boolean vacant = false;
if (vacancy.as_int()>0) vacant = true;
@send(agency,reply.post(ask.ch,ask.tv,ask.sm,ask.bu,ask.pri,ask.len,vacant));
```

7.3.4 A database : TicketsCompDB

The database code contains the fields of the database and the specification of the methods that allow accessing the data.

Declaration of the database "TicketsCompDB"

"*TicketsCompDB*" is the database of the CompanyAgent that contains all its tickets. "*TicketCompDB*" represents a "*ClosedWorld*" relation. A "*ClosedWorld*" relation assumes that the agent is operating in a closed world. This means that it assumes that every tuple the relation can express is stored in the database as being either true or false at all times. On the contrary, "*OpenWorld*" relations model knowledge and beliefs as they are experienced by

most people in the real world : some things may be known to be true, others known to be false and still others unknown.

```
database TicketsCompDB extends ClosedWorld {
```

Fields' declaration

The different fields of the beliefs that will be stored in the database are declared as follows. "*Index*" is a "*key field*", i.e. it uniquely identifies the object or entity to which the tuple refers.

```
#key field int index;
#value field String departure;
#value field String destination;
#value field String champ;
#value field String TV;
#value field String smoker;
#value field String business;
#value field int price;
#value field double length;
#value field int vacancy;
```

Queries

Once a database relation has been defined and tuples have been added to the database, the agent will need to access these data. It does so by performing a query on the relation. The four following "*#indexed query*" statements describe the way queries to the database are executed. Their parameters are either defined as normal members (i.e. input parameters) or logical members (i.e. output members that have to be unified). For example, the first of these statements indicates that the database can be queried by specifying only the number (*i*) of the tuple in the index. The parameters defined as logical members will then be unified with the values in the fields of that tuple.

```
#indexed query get(int i, logical String dep, logical String dest,logical String c, logical String t ,
logical String s, logical String b, logical int p, logical double h, logical int v);

#indexed query get(logical int i, String dep, String dest,logical String c, logical String t , logical
String s, logical String b, logical int p, logical double h, logical int v);

#indexed query get(logical int i, String dep, String dest, String c, String t , String s, String b,int
p,double h, logical int v);

#indexed query get(logical int i, String dep, String dest, String c, String t , String s, String
b,logical int p,logical double h, logical int v);
}
```

7.3.5 A capability : DefCap

The code of a capability looks like an agent. The code declares databases the agent use, which events it sends or receives and which plans it can execute. The code contains also the methods of the capability.

Capability declaration

"*DefCap*" is a capability of the TravelAgent. It gathers the plans that implement the default reasoning.

```
capability DefCap extends Capability {
```

Java statements

First, we define a DefaultTheory (the data structure of Hades, see chapter 5) that is specific to this capability.

```
public DefaultTheory dt = new DefaultTheory();
```

Jack statements (events, plan and databases)

As in the *Agent*'s definition, we have to declare the **events** the capability will treat when they will arise and the corresponding **plans**.

```
#handles external event DefaultE;
#handles event InitDTE;
#handles event CalculateExtE;

#uses plan DefaultP;
#uses plan InitDTP;
#uses plan CalculateExtP;
```

Next, we have to specify the **databases** of the agent that will be accessed in the plans of the capability.

```
#imports database RulesDB dbRules();
#imports database TicketsDB dbTickets();
#imports database TicketsDB2 dbTickets2();
#imports database DefaultsDB dbDefaults();
```

Methods

And finally, we define two methods of the capability. These are a method to access the Default Theory and a method to assign it a content.

```
public DefaultTheory getTheory()
{ return dt; }

public void setTheory(DefaultTheory dtRules)
{ dt=dtRules; }
}
```

8.Conclusion

When we look back at our research experience, we can say that building an agent with the existing tools can be achieved without too much difficulty. Indeed, we found that Jack, the agent-oriented programming language we used in our application, is not much more complicated than a classical language such as Pascal or C. However, we are aware of the fact that building complex and efficient distributed systems is never easy but that an agent-oriented approach allows to tackle such systems with more ease.

Integrating more complex modules based on Artificial Intelligence, and non-monotonic reasoning obviously may complicate the process. Thanks to the tools currently available, it is made possible to build an agent, or even a multi-agent system. In our application, we encountered not that many difficulties in integrating functionalities based on non-monotonic logic into our agent. Maybe the most difficult thing was finding domains of application where Artificial Intelligence could help Intelligent Agents to act more efficiently. It took us quite some time to find out how Belief Revision can be adapted to the TravelAgent reasoning.

Apparently, the Intelligent Agent technology is already well present, but most of the agents developed still have more or less basic reasoning methods. Most of them are closer to software agents than to Intelligent Agents. What we tried to do was including Default Logic and Belief Revision in the reasoning methods of our agents and analysing the efficiency of these techniques.

Unfortunately, we encountered one of the problems of AI-based Intelligent Agents, that is the computational power it may require. The aim of using a belief set and a system of selection based on Belief Revision was to copy the human methods of reasoning. But, because of the slowness of the tools we used to implement the non-monotonic logic modules, the processing times can be very long, event though the application is working with very small databases. Nevertheless, this could be powerful some day, but given the limits of the current computational power, our system is unusable because of its slowness. Our application is just a prototype, we did not intend to build a ready-to-use Intelligent Agent. With more efficient tools at our disposition, the problem could be solved. For example, time of processing could easily be gained by using logical programming languages such as Prolog.

GENERAL CONCLUSION

In the **first part** of our dissertation, we discussed the theoretical basis of Intelligent Agents' technology and non-monotonic logic. The aim of this part is to understand the main concepts related to the exciting field of Intelligent Agents and to understand an AI technique (non-monotonic logic) that can be used to give a sense to the word intelligent in "Intelligent Agent".

This part shows that there is a great diversity in agents research and, therefore, in the notion of agent. However some consensus can be found and several concepts related to Intelligent Agents are now well defined. Particularly, elegant solutions have been found for issues such as agents' architectures, agents' communication and agents' languages. Some candidate standards appear also but are not yet widely used.

Concerning non-monotonic logic, we present the theoretical aspects of this technique. Non-monotonic concepts are now well understood and defined by the AI community. It provides a way to imitate human reasoning better and better. It remains now to build real applications using among others the agent paradigm.

In the **second part**, we state that it is currently possible to build an agent-based application using agent-oriented tools and AI techniques. The objective of this part is to analyse the needs of the application, then to design a framework using agent concepts and finally to build the application using specific tools.

Our application proves that agents can be a significant technology for software engineering. This implies that AI is not central any more for the agent paradigm, although the idea of agent comes from the AI community. The word agent knows a large acceptation nowadays and it is currently possible to build agents without using AI techniques. In these cases, it is more appropriate to use the term "software agent" than Intelligent Agent. AI may always give a sense to "intelligent" in Intelligent Agent and the agent paradigm remains an attractive field to build AI applications.

We demonstrate that the agent concept is nearly at a mature stage of development and that the widespread acceptance of the concept implies that agents have a significant future. Now that most concepts are defined, it is time to address the practical issues surrounding the development of systems that use the agent technology. Research will be forced to tackle real problems. Therefore, it is interesting to build real applications (as we did in part two) in order to find a balance between theoretical concepts and practical issues.

Indeed, a rift may appear between commercial, simple and efficient Intelligent Agents in which the public is really interested and more complex AI-based agents that do not always fulfil real needs. Users will be the ultimate test of an agent's success. They will also drive Intelligent Agents' development. This is something that seems to be certain. What is uncertain is whether users will discover, use and adopt agents all by themselves, or whether they will just start using them because they are incorporated into a majority of applications.

There exists already evidence of greater market sector segmentation into clear application focused areas. Where there are financial imperatives, the research will follow. However, it seems to be important to have a better understanding of the situations in which agent solutions are appropriate because the temptation to see agents everywhere is big.

Anyway, from the current situation it cannot be easily deduced which path future developments will follow. There is not yet massive supply of agents or agent-based applications, but what can be seen is that large software and hardware companies, such as IBM, Microsoft and Sun Microsystems, are busy studying and developing agents (or agent-like techniques) and applications.

Intelligent Agents won't take a long time to make themselves known and soon they will be indispensable, especially on the Internet. The growing popularity of the Internet, but also the problems many people encounter when searching for or when offering information or services on it, will only increase the possible number of applications or application areas : the Internet is an ideal environment for agents as they are (or can be) well adapted to its uncertainty, and are better at dealing with the Internet's complexity and extensiveness.

Intelligent Agents will be the compulsory intermediaries to work on the Internet and AI will maybe help them to act as 'intelligently' as a human would do. In a further future, we can imagine that every company and maybe every individual will be represented on the Internet by his or her Intelligent Agent that, thanks to AI-based reasoning methods, would behave really 'intelligently', e.g. they would learn as much as possible of who its owner is : his/her habits, his/her knowledge, his/her way of communication, his/her circle of relations, ... All these agents will complete the landscape of the Internet to form the virtual world of information. There already exist the virtual highways of information. Now, one can add agents to put on them. This helps the human kind in its search of speeding up the treatment of information even more. It is already a good thing to have the highways of information but their advantages cannot be fully appreciated if human agents (who are too slow for this virtual *speedy* world) are driving on them.

The agent-technique is still very young. It will take a lot of trial and error, and a lot of experimenting to make it mature. This is exactly the stage in which we are right now, and thus we cannot expect agents to be already advanced and (nearly) perfect. Developments may not have achieved perfect agents yet, but they certainly have made enough progress to make agents more than just a hype.

BIBLIOGRAPHY

Books

- ANTONIOU (G.), *Non-monotonic Reasoning*, ed. The MIT Press, Cambridge, Massachusetts, London, England, 1997.

- BRADSHAW (J.), *Software Agents*, AAAI Press/The MIT Press, 1997. Introduction available at http://www.cs.umbc.edu/agents/introduction/01-Bradshaw.pdf

- GÄRDENFORS (P.) (edited by), *Belief Revision*, in Cambridge Tracts in Theoretical Computer Science, ed. Cambridge University Press, n°29, 1992.

- GRÉGOIRE (E.), *Logiques non-monotones et intelligence artificielle*, coll. Langue, Raisonnement, Calcul, ed. Hermes, Paris, 1990.

- MAREK (V.W.), TRUSZCZYNSKI (M.), *Non-monotonic Logic, Context-Dependant Reasoning*, Springer-Verlag, Berlin Heidelberg 1993.

- MÜLLER (J.P.), *The Design of Intelligent Agents : A Layered Approach*, Lecture Notes in Artificial Intelligence, 1996, Springer.

- NWANA (H.), AZARMI (N.), *Software Agents and Soft Computing : Towards Enhancing Machine Intelligence,* Lecture Notes in Artificial Intelligence, 1998, Springer.

- WEISS (G.), SEN (S.) , *Adaptation and Learning in Multi-Agent Systems*, Lecture Notes in Artificial Intelligence, 1996, Springer-Verlag.

- WOOLDRIDGE (M.), JENNINGS (N.), *Agent Theories, Architectures, and Languages*, Lecture Notes in Artificial Intelligence, 1996, Springer.

Papers

- BASAWARAJ (P.), "Agents and Agent technologies : An overview", 1998. Available at http://www.darmstadt.gmd.de/~patil/agent_report.html

- BERNEY (B.), "Software Agents : A review", Manchester. Available at http://www.doc.mmu.ac.uk/STAFF/B.Berney/research/ag-rev.htm

- BLOCH (M.), SEGEV (A.), "The Impact of Electronic Commerce on the Travel Industry, An Analysis Methodology and Case Study", Berkeley, 1996. Available at http://www.stern.nyu.edu/~mbloch/docs/travel/travel.htm

- BUSETTA (P.), KOTAGIRI (R.), "An Architecture for mobile BDI Agents in Applied computing", in *Symposium on applied computing (SAC 98)*, 1998, pp 445-452. Available at http://www.cs.mu.OZ.AU/~paolo/TOMAS/sac98.ps

- BUSETTA (P.), RÖNNQUIST (R.), HODGSON (A.), LUCAS (A.), "JACK Intelligent Agents Components for Intelligent Agents in Java", AgentLink News Letter, 1999. Available at http://www.agent-software.com/whitepaper/html/whitepaper.html

- COBURN (M.), "JACK User Guide Agent Oriented Software Pty. Ltd.", release 2.1, 2000. Available at http://www.agent-software.com/docs/jack/html/

- DICKISON (I.), "Agent Standards", in *Helwett-Packard Technical Report HPL-97-156*, Hewlett-Packard Laboratories, Bristol. Available at http://agents.hpl.hp.com/papers/actcc%20paper.ps

- DO (O.), MARCH (E.), RICH (J.) and WOLFF (T.), "Intelligent Agents & the Internet, Effects on electronic commerce and marketing". Available at http://mrmac-jr.scs.unr.edu/odie/paper.htm

- FALKENROTH (E.) and GRANLUND (R.), "Notes in Intelligent Software Agents : Introduction", courses held by the Laboratory for Intelligent Information Systems (IISLAB) Linköpings universitet , 1998. Available at http://www.ida.liu.se/labs/iislab/courses/Agents/paper/chapter1.html

- FAQUHAR (A.), FIKES (R.) and RICE (J.), "The ontolinguia Server : a tool for collaborative Ontology Construction". Available at http://ontolinguia.stanford.edu/

- FININ (T.), LABROU (Y.), Tutorial on "Agent Communication Languages", University of MaryLand Baltimore County, First International Symposium on Agent Systems and Applications and the Third International Symposium on Mobile Agents, California, U.S.A, 1999. Available at http://www.csee.umbc.edu/~finin/papers/asama99tutorial.shtml

- FLORES-MENDEZ (R.), "Towards a standardisation of Multi-Agent System Frameworks", 2000. Available at http://www.acm.org/crossroads/xrds5-4/multiagent.html

- FRANKLIN (S.), GRAESSER (A.), "Is it an Agent, or just a Program? : A Taxonomy for Autonomous Agents", in *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Language,* Springer-Verlag, 1996. Available at http://www.eurecom.fr/~diana/bib/FG96/main.html

- GENESERETH (M.), KETCHPEL (S.), *Software Agents*. Communications of the ACM, 37 (7), July 1994, pp48-53. Available at http://ai.about.com/compute/ai/gi/dynamic/offsite.htm?site=http://logic.stanford.edu/sharing/papers/agents.ps

- GILBERT and APARICIO, "The Role of Intelligent Agents in the Information Infrastructure", IBM, United States, 1995. Available at http://activist.gpl.ibm.com:81/WhitePaper/ptc2.htm

- GRUBER (T.), "What is an ontology?", 1993 . Available at http://www-ksl.stanford.edu/kst/what-is-an-ontology.html

- HAYZELDEN (A.), BIGHAM (J.), "Agent Technology in Communication Systems : An overview", in *Knowledge Engineering Review*, London, 1998. Available at http://agents.umbc.edu/cgi-bin/raw?url=http://www.agentcom.org/papers/ker-99.pdf

- HEILMANN (K.), KIHANYA (D.), LIGHT (A.) and MUSEMBWA (P.), "Intelligent Agents : A technology and business application analysis", 1995. Available at http://www.mines.u-nancy.fr/~gueniffe/CoursEMN/I31/heilmann/heilmann.html

- HENDLER (J.), "Is There an Intelligent Agent in Your Future?", Macmillan Publishers Ltd 1999. Available at http://helix.nature.com/webmatters/agents/agents.html

- HERMANS (B.), "Intelligent Software Agents on the Internet : An Inventory of Currently Offered Functionality in the Information Society and a Prediction of (near) Future Developments", 1997. Available at http://www.firstmonday.dk/issues/issue2_3/ch_123/

- LABROU (Y.), FININ (T.) and PENG (Y.), "The current landscape of Agent Communication Languages" in *Intelligent Systems*, Vol. 14, No. 2, IEEE Computer Society, March/April 1999. Available at http://www.cs.umbc.edu/~jklabrou/publications/ieeeIntelligentSystems1999.pdf

- LIU (W.), WILLIAMS (M.-A.), "A framework for Multi-Agent Belief Revision, Part II: A Layered Model and Shared Knowledge Structure", in *Linköping Electronic Articles in Computer and Information Science*. Available at http://www.ida.liu.se/ext/epa/cis/ufn-00/03/tcover.html

- MAES (P.), GUTTMAN (R.) and MOUKAS (A.), "Agents that Buy and Sell: Transforming Commerce as we Know It", Communications of the ACM, 1999, Cambridge. Available at http://mas.cs.umass.edu/~aseltine/791S/maes.agents_that_buy.pdf

- MCCABE (F.G.), "Liaison Proposal OMG-FIPA", OMG Document # liaison/93-03-03. Available at http://www.objs.com/isig/omg-fipa-liaison-4.html

- MICHAELIDES (A.), MORAITAKIS (N.) and GEORGALIDES (G.), "Intelligent Software Agents in Travel Reservation Systems", London, 1997. Available at http://www-dse.doc.ic.ac.uk/~nd/surprise_97/journal/vol4/nm1

- NATHAN (P.X.), GARNER (R.G.), "The evolution of Intelligent Agents on the Web", 1997. Available at http://www.robitron.com/paper1_5.html

- NWANA (H.), "Software Agent : An overview", in *TheKnowledge Engineering Review Journal*, vol. II, n°3, pp.1-40, Cambridge University Press, 1996. Available at http://193.113.209.147/projects/agents/publish/papers/agentreview.htm

- NWANA (H.), NDUMU (D.), "A perspective on Software Agent Research", in *The Knowledge Engineering Review*, Vol 14, No 2, Ipswitch, 1999. Available at http://agents.umbc.edu/introduction/hn-dn-ker99.html

- NWANA (H.), WOOLDRIDGE (M.), "Software Agent Technologies", 1996. Available at http://www.labs.bt.com/projects/agents/publish/papers/sat_report.html

- OBJECT MANAGEMENT GROUP, "Agent Technology Green Paper", Framingham, 2000. Available at www.iti.upv.es/iti/i+d/mirrors/ftp.omg.org/pub/docs/ec/00-04-01.txt

- O'BRIEN (P.) (edited by), "Agent Management", in *Agent Management FIPA '98 Draft Specification : Part1*, FIPA-Foundation for Intelligent Physical Agents, Geneva, Switzerland, 1998. Available at http://www.cselt.it/fipa/spec/fipa98/fipa98.htm

- RAO (A.S.) AND GEORGEFF (M.P.), "BDI agents: From theory to practice", Technical Report 56, Australian Artificial Intelligence Institute, Melbourne, Australia, 1995.

- VERSTEEG (S.), STERLING (L.). "Languages for mobile agents", Thesis at Melbourne University, 1997. Available at http://www.cs.mu.oz.au/~scv/thesis.html

- WILLIAMS (M-A.), WILLIAMS (D.), "A Belief Revision System For the World Web", Newcastle, Australia. Available at http://u2.newcastle.edu.au/webworld/ai-internet.html

- WOOLDRIDGE (M.), "Intelligent Agents", in *Multiagent Systems*, ed. by Weiss, MIT Press Cambridge, Massachussetts, 1999, pp. 27-77.

- WOOLDRIDGE (M.), JENNINGS (N.), "Agent-Oriented Software Engineering", in *Handbook of Agent Technology*, ed. J. Bradshaw AAAI/MIT Press, 2000. Available at http://www.ecs.soton.ac.uk/~nrj/download-files/agt-handbook.pdf

Annexes

List of Annexes

Annex 1 : ***Description of the content of the plans of TravelAgent and CompanyAgent***

The TravelAgent's plans

The DefCap capability's plans

The DefCap capability treats default reasoning. It contains 3 plans : *DefaultP, InitDTP* and *CalculateExtP*.

DefaultP

DefaultP is a kind of coordinator of the capability. It triggers off the 2 other plans (by posting the adequate events).

InitDTP

InitDTP initializes the Default Theory, which is the data structure defined in HADES. It is composed of an array of *strings* (*in*) for the beliefs (or facts), an array of *default* objects (*defaults*) for the defaults and an array of arrays of *String* (*extensions*) for the extensions. The class *default* is composed of a *pre* (a *string* representing the precondition or prerequisite), a *justs* (an array of *strings* representing the justifications) and a *cons* (a *string* representing the consequent).

First, it creates a new Default Theory.

```
dtRules = new DefaultTheory();
```

Than, it fills the Default Theory with rules representing the information about the tickets transformed (by the *TreatData* capability) into rules in order for the extensions to be calculated.

For example, if the first ticket ('ticket1') is a business class ticket, there will be a rule :

ticket1 -> business

Here is how it fills the Default Theory with these rules :

```
for(int i=1;i<=dbRules.nFacts();i++)
        {
        logical String str ;
        dbRules.get(i,str);
        if (str.as_string().indexOf("->")!=-1)
                {
                dtRules.addFact(str.as_string());
                }
        }
```

It than fills the Default Theory with the default rules (that were stored in the database *dbDefaults*). Remember that a default rule is a data structure composed of a premise, justifications and a consequent.

```
for(int j=1;j<=dbDefaults.nFacts();j++)
        {
```

```
logical String s1,s2,s3;
Default d = new Default();
dbDefaults.get(j,s1,s2,s3);
d.pre = s1.as_string();
d.justs.append(s2.as_string());
d.cons = s3.as_string();
dtRules.addDefault(d);
}
```

ClaculateExtP

This is the plan that calculates the extensions.

It generates the extensions and store them in an array of arrays of strings. dtRules is the Default Theory that was created in the plan *InitDTP*.

```
String_Array_Array extens = new String_Array_Array();
extens = dtRules.generate();
```

It inserts in the database *dbRules* those formulas of the extensions that are not in *dbRules* yet.

```
For (int l=0; l<extens.mod();++l)
    {
    for (int j=1;j<=extens.at(l).mod();++j)
        {
        logical int ind;
        Cursor presence = (dbRules.get(ind,extens.at(l).at(j-1)));
        if (!presence)
        {
        dbRules.assert(dbRules.nFacts()+1,extens.at(l).at(j-1));
    }
```

The Br capability's plans

This capability contains the plans *AnalyseTBP, InitRulesP, InitCriteriasP, ChgeCriteriasP, ExtractP, RemoveTicketP* and *ReviseP*.

InitRules

This is a plan that initializes the TheoryBase, i.e. that fills it with the beliefs. In our TheoryBase, the beliefs are the criteria's (stored in the *dbCriteres2*) which represent the preferences of the user and the rules (stored in the *dbRules*) which represent the characteristics of the tickets. This plans has to fill the TheoryBase with the rules.
When an element is added in the TheoryBase, it must be assigned a rank (this is the second argument of the method addBelief). This rank depends on the type of rule. The OR-formula will be assigned the highest rank (0.9999). The rules of type 'ticket1->champagne' will receive a rank a little bit weaker (0.9998), but stronger than the ranks of the criteria's (see *InitCriteriasP*). And the simple rules of type 'ticket2' will have the lowest rank (0.0001).

```
TheoryBase tb = cap.getTheory()
```

```
for (int i=1; i<=dbRules.nFacts(); i++)
 {
        logical String rule;
        dbRules.get(i,rule);
        if (!tb.contains(rule.as_string()))
                {
                if (rule.as_string().length()==7)  // EX : ticket2
                        {
                        tb.addBelief(rule.as_string(),0.0001);
                        }
                if (rule.as_string().indexOf("|")!= -1)  // EX : ticket1|ticket2
                        {
                         tb.addBelief(rule.as_string(),0.9999);
                        }
                if (rule.as_string().indexOf(">")!= -1)  // EX : ticket1->champagne
                        {
                        tb.addBelief(rule.as_string(),0.9998);
                        }
                }
 }
```

InitCriterias

This is another plan that initializes the TheoryBase. Whereas *InitRules* fills the TheoryBase with the rules, *InitCriterias* fills it with the criteria. These criteria's have the form : champagne0.5. The ranking assigned depends on the preferences expressed by the user. They are higher than 0.0001 and lower than 0.9998.

```
TheoryBase tb = cap.getTheory();
for (int i=1; i<=dbCriteres2.nFacts(); i++)
        {
                logical String belief;
                logical double ranking;
                dbCriteres2.get(i, belief, ranking);
                tb.addBelief(belief.as_string(),ranking.as_double());
        }
```

It also posts the event that will trigger off the plan *ExtractP*.

```
@subtask(extr.post());
```

Finally, it analyses the TheoryBase to get the remaining ticket.

```
@subtask(anal.post());
```

ExtractP

This is the plan that extracts a consistent theory from the TheoryBase. It uses the object Extractor of SATEN in order to achieve its goal. *ExtractP* is used at the first search for tickets, i.e. when the agent has to fill its TheoryBase with all the preferences of the users and all the

characteristics of the different tickets. It is clear that it wouldn't be adequate to revise (i.e. inserting + extracting) the TheoryBase with each of the new elements inserted. On the contrary, if later the user changes his mind about his preferences or if a characteristic of one ticket changes, the TheoryBase can be revised with the change. The *ReviseP* would be suitable for that kind of situation.

```
TheoryBase tb = cap.getTheory();
getExtractor(tb).extractTheory(tb);
```

tb is the TheoryBase. GetExtractor is a *reasoning method* (a kind of procedure in Jack) that will orient the extraction to one of the possible Extractor (Standard Adjustment, Maxi Adjustment, Hybrid Adjustment, ...). In our application, we will always use Maxi Adjustment.

ReviseP

This is the same plan as the plan *ExtractP* except that it uses the method *revise* of the chosen Extractor instead of the method *extract*.

```
TheoryBase tb = cap.getTheory();
getExtractor(tb).revise(ev.at,ev.rank);
```

AnalyseTBP

Once the extraction is made, one has to analyze the TheoryBase to see what the remaining ticket (i.e. the best ticket) is and what its characteristics are.

```
TheoryBase tb = cap.getTheory();
```

It retrieves the number of the best ticket which is normally in last position in the TheoryBase.

```
String tic = tb.beliefs.at(tb.beliefs.mod()-1);
int numTicket = Integer.valueOf(tic.substring(6,7)).intValue();
```

All the information about the ticket is not explicitly in the TheoryBase. The agent has to deduce the negative attributes (which are not explicitly in the TheoryBase after the extraction).

Finally, the plan posts the event *BookingE* that will trigger off the plan *BookingP*.

```
@post(book.post());
```

RemoveTicketP

When there is no vacancy for the best ticket found, the agent has to remove this ticket and spawn the search again to find the second best ticket.

```
TheoryBase tb = cap.getTheory();
```

Here is how the plan takes the number of the best ticket that will be removed

```
logical int numTicket,numPass;
logical String str;
TicketBest.get(TicketBest.nFacts(),numTicket,numPass,str);
```

Removing the best ticket means rewriting the OR-formula (which is at the first position in the TheoryBase) without this ticket.

```
tb.beliefs.stableRemove(0);
tb.rankings.stableRemove(0);

String disjunction = "";
for (int j=1;j<=dbTickets2.nFacts();j++)
{
        logical int ind,pass;
        logical String str;
        if (!(TicketBest.get(ind,j,pass,str))) disjunction=disjunction + "ticket"+j+"|";
}
tb.beliefs.setAt(disjunction,0);
tb.rankings.setAt(0.9999,0);
```

Now the best ticket has been removed, one can post the event *ExtractE* to spawn the extraction …

```
@subtask(extr.post());
```

… and post the event *AnalyseTBE* to start the analysis of the TheoryBase that will single out the new best ticket.

```
@subtask(anal.post());
```

ChgeCriteriasP

This plan is used when the user decides to change his preferences and spawn a new search. Every new criteria (stored in the database *dbCriterias*) is moved in the TheoryBase according to its new ranking.

```
TheoryBase tb = cap.getTheory();

for (int i=1; i<=dbCriterias.nFacts(); i++)
        {
        logical String belief;
        logical double ranking;
        dbCriterias.get(i, belief, ranking);
        tb.movBelief(belief.as_string(), ranking.as_double());
        }
```

Than, as usual, one posts the plan *ExtractP* to spawn the extraction

```
@subtask(extr.post());
```

And finally, we spawn the analysis of the TheoryBase to retrieve the best ticket and its characteristics.

```
@subtask(analys.post());
```

The BookingCap Capability's plans

This capability is used to book –if some vacancy is found- the best ticket. It contains 2 plans : *BookingP* and *ComReplyVacancyP*.

BookingP

The plan has to find which company issued the best ticket found. In order to achieve that, it searches in the database dbTicket2 which company corresponds to the characteristics of the ticket.

```
logical int i;
logical String company;
dbTickets2.get(i,champ,tv,smo,bus,price,length,company);
```

Once the company is found, the agent asks it - via the communication capability - for vacancy

```
@post(comm.post(company.as_string(),dep.as_string(),dest.as_string(),
champ.as_string(),tv.as_string(),smo.as_string(),bus.as_string(),price.as_int(),
length.as_double()));
```

ComReplyVacancyP

This plan has to treat the response of the company that has been asked whether or not there was vacancy for the chosen ticket.

If its answer is positive, the ticket is booked,

```
dbBooked.assert(dbBooked.nFacts()+1,ch.as_string(),tv.as_string(),
sm.as_string(), bu.as_string(),comm2.p,comm2.l,comm2.co);
```

On the contrary, if there is no vacancy, the ticket is queued. It means that the agent is candidate for a ticket that could be freed by the cancellation of a booking.

```
dbQueued.assert(dbQueued.nFacts()+1,ch.as_string(),tv.as_string(),sm.as_string(),
bu.as_string(),comm2.p,comm2.l,comm2.co);
```

In that case, another search is spawned –via the plan *RemoveTicketP*- to look for another ticket (because waiting for the cancellation of a booking is too hazardous).

```
@subtask(remove.post());
```

The TreatDataCap Capability's plans

The *TreatDataCap* capability cares for the format of the formulas that will be inserted in the TheoryBase. It contains 3 plans : *TransformationP*, *TreatCrtiterP* and *TreatTicketsP*.

TransformationP

It creates, with each characteristic of a ticket, a rule of the form 'ticket1->business' and inserts it in the database *dbRules*.

```
for(int i=1;i<=dbTickets2.nFacts();i++)
        {
        logical String ch,t,sm,bu,pri,len,comp;
        dbTickets2.get(i,ch,t,sm,bu,pri,len,comp);

        if (ch.as_string().length()!=0 )dbRules.assert(dbRules.nFacts()+1,"ticket" + i
        + "->" + ch.as_string());
        if (t.as_string().length()!=0 )dbRules.assert(dbRules.nFacts()+1,"ticket" + i +
        "->" + t.as_string());
        if (sm.as_string().length()!=0 )dbRules.assert(dbRules.nFacts()+1,"ticket" + i
        + "->" + sm.as_string());
        if (bu.as_string().length()!=0 )dbRules.assert(dbRules.nFacts()+1,"ticket" + i
        + "->" + bu.as_string());
        }
```

In the case of the price and the length, which are quantitative adjectives, another treatment has to be done. This has been explained in a previous section.
It also forms the OR-formula, based on the number of tickets proposed. For example, if the user is looking for a flight from Brussels to Sydney on the first of January, and if 4 flights are proposed by the different companies, the OR-formula would be 'ticket1|ticket2|ticket3|ticket4'

It than adds the formed formula in *dbRules*.

```
dbRules.assert(dbRules.nFacts()+1,disjunction);
```

And, finally, it inserts the names of the different tickets in *dbRules*.

```
for (int j=1;j<=dbTickets2.nFacts();j++)
        {
        dbRules.assert(dbRules.nFacts()+1,"ticket"+j);
        }
```

TreatCriterP

This plan transforms the quantitative criteria's into 5 derived criteria's (see the 'How does our TheoryBase work' section). *dbCriteres* contains the 'pure' criteria's while *dbCriteres2* contains the transformed criteria's. The non quantitative criteria's are just recopied from *dbCriteres* to *dbCriteres2*.

```
for (int i=1;i<=dbCriteres.nFacts();i++)
        {
        logical String name;
```

```
logical double rank;
dbCriteres.get(i,name,rank);

if (name.as_string()=="price" | name.as_string()=="length")
        // quantitative criteria's
        {
        dbCriteres2.assert(dbCriteres2.nFacts()+1,name.as_string()+
        "1",(1.0)*rank.as_double());
        dbCriteres2.assert(dbCriteres2.nFacts()+1,name.as_string()+
        "2",(0.8)*rank.as_double());
        dbCriteres2.assert(dbCriteres2.nFacts()+1,name.as_string()+
        "3",(0.6)*rank.as_double());
        dbCriteres2.assert(dbCriteres2.nFacts()+1,name.as_string()+
        "4",(0.4)*rank.as_double());
        dbCriteres2.assert(dbCriteres2.nFacts()+1,name.as_string()+
        "5",(0.2)*rank.as_double());
        }
        // non quantitative criteria's
else dbCriteres2.assert(dbCriteres2.nFacts()+1,name.as_string() ,
rank.as_double());
}
```

For example, if the content of *dbCriteres* is :

champagne 0.9
price 0.8
smoker 0.2

Then *dbCriteres2* will be filled with :

champagne 0.9
price1 0.8
price2 0.64
price3 0.48
price4 0.32
smoker 0.2
price5 0.16

TreatTicketsP

This plan delimitates the intervals for the quantitative characteristics of the tickets (i.e. the price of the tickets and the length of the flight) in order to transform them into boolean criteria's (see the 'How do we use belief revision ?' at section 6.1.2. ?????). First, it calculates the higher and lower bounds among the proposed tickets. Here is the code corresponding to the treatment of the price.

```
int bestPrice = 99999;
int worstPrice = 0;

for (int i=1; i<=dbTickets.nFacts();i++)
        {
        logical String s1,s2,s3,s4;
        logical int price;
```

```
logical String company;

dbTickets.get(i,s1,s2,s3,s4,price,length, company);
if (price.as_int() <= bestPrice) bestPrice = price.as_int();
if (price.as_int() >= worstPrice) worstPrice = price.as_int();
}
```

Than, it divides the space between the lower and higher bounds into 5 intervals of the same size.

```
int priceBound12 = bestPrice + (worstPrice - bestPrice)/8;
int priceBound23 = priceBound12 + (worstPrice - bestPrice)/4;
int priceBound34 = priceBound23 + (worstPrice - bestPrice)/4;
int priceBound45 = priceBound34 + (worstPrice - bestPrice)/4;
```

And finally, it classifies each ticket into one of the intervals and gives it the corresponding attribute.

```
for (int j=1; j<=dbTickets.nFacts();j++)
        {
        logical String s1,s2,s3,s4;
        logical int price;
        logical double length;
        logical String company;

        String priceString;

        dbTickets.get(j,s1,s2,s3,s4,price,length,company);

        if (price.as_int() <= priceBound12) priceString = "price1";
        if (price.as_int() >= priceBound12
                & price.as_int() <= priceBound23 ) priceString = "price2";
        if (price.as_int() >= priceBound23
                & price.as_int() <= priceBound34 ) priceString = "price3";
        if (price.as_int() >= priceBound34
                & price.as_int() <= priceBound45 ) priceString = "price4";
        if (price.as_int() >= priceBound45) priceString = "price5";


        dbTickets2.assert(j,s1.as_string(),s2.as_string(),s3.as_string(),s4.as_string(),
                        priceString, lengthString, company.as_string());
        }
```

The Communication capability's plans

The Communication capability gathers all the plans of the TravelAgent that are involved in the communication between the TravelAgents and the CompanyAgents. It contains the following plans : *ChgeGenCriteriasP, ReceiveTicketsP, AllTicketsP, ComAskVacancyP, ReplyVacancyP* and *ChgeTicketP.*

ChgeGenCriteriasP

It is the first plan used in the 'call for tickets' process. It sends the asking for tickets (corresponding to the general criteria's) to the different CompanyAgents.

```
logical String dep, dest;
dbGeneralCriterias.get(1,dep,dest);

for (int i=1; i<=dbSociale.nFacts(); i++)
        {
        logical String company;
        dbSociale.get(i, company);
        @send(company.as_string(),ask.asked(dep.as_string(),dest.as_string()));
        }
```

Than, the CompanyAgents reply by sending the tickets they propose and a 'closure' message. The tickets are treated by the *ReceiveTicketsP* and the closure messages by the *AllTicketsP*.

ReceiveTicketsP

The CompanyAgents send their tickets in the MessageEvent *ReceiveTicketsE*. So, for each ticket sent, a plan *ReceiveTicketsP* is triggered off. First, the plan extracts the name of the company from the address of the sender of the message (containing the ticket).

```
int ind = tic.from.indexOf("@");
String company = tic.from.substring(0,ind);
```

Indeed, it needs the name of the company that issued the ticket to store it in the database *dbTickets* (that will contain all the tickets among which the best ticket will be singled out).

```
dbTickets.assert(dbTickets.nFacts()+1,tic.ch,tic.tv,tic.sm,tic.bu,tic.pr,tic.le,company);
```

AllTicketsP

Each time a 'closure' message is sent to the TravelAgent, the plan *AllTicketsP* is executed. The plan has to count the number of messages received to know whether all the CompanyAgents have sent all their tickets. In order to count the number of occurrences of the plan (i.e. the number of *AllTicketsE* sent) , we need a *static* counter, so that it won't be reinitialized each time the plan is executed.

```
static int i=0 ;
```

The first instruction of the body of the plan is incrementing that counter.

```
i++ ;
```

If the plan has been executed as many times as there are CompanyAgents, the treatment of the tickets received can begin.

```
if (i==3)
{
        @subtask(data.fillDBs());
        @subtask(tick.classif());
        @subtask(tra.transf());
        @subtask(defa.def());
        @subtask(initR.post());
        @subtask(initC.post());
}
```

ComAskVacancyP

This plan is executed when the best ticket is found and the TravelAgent asks the concerned CompanyAgent if there are seats available for the flight that corresponds to that ticket. The asking has the form of a MessageEvent containing all the characteristics of the tickets, so that the CompanyAgent can retrieve it in its database.

```
@send(comm.com,ask.post(comm.dep,comm.dest,comm.ch,comm.tv,comm.sm,
        comm.bu,comm.pri,comm.len));
```

ComReplyVacancy

This is just the plan that is in charge with receiving the answer of the CompanyAgent concerning the question of vacancy. If this answer is positive, the ticket is booked.

```
dbBooked.assert(dbBooked.nFacts()+1,ch.as_string(),tv.as_string(),sm.as_string(),
        bu.as_string(),comm2.p,comm2.l,comm2.co);
```

If it is negative, the ticket is queued...

```
dbQueued.assert(dbQueued.nFacts()+1,ch.as_string(),tv.as_string(),sm.as_string(),
        bu.as_string(),comm2.p,comm2.l,comm2.co);
```

... and another search without the ticket that has been queued is spawned.

```
@subtask(remove.post());
```

ChangeTicketsP

ChangeTicketsP handles the events *ChangeTicketsE* sent by a CompanyAgent. CompanyAgents send such messages when a change occurs in their database. These agents keep track of the kind of tickets that the TravelAgents have asked them. Hence, if for example, the price of a ticket for a flight from Brussels to Paris changes, only the TravelAgents that have asked tickets from Brussels to Paris will be warned. In fact, the change in the CompanyAgents' database can be of three types : a change of characteristics of a ticket (an update), the removal of a ticket or the apparition of a new ticket.

If the operation is an update, the MessageEvent *ChangeTicketsE* contains all the new characteristics of the ticket as well as all its old characteristics. These old characteristics are

necessary for the TravelAgent to find in its database the track of the ticket that has to be updated. If the operation is an addition of a new ticket, all the fields representing the old characteristics are set to "" while if it is a removal of a ticket, these are the new characteristics that will be set to "".

First, it has to extract the name of the company from the address of the agent that sent the MessageEvent .

```
int ind = change.from.indexOf("@");
String company = change.from.substring(0,ind);
```

With the name of the company found coupled with the old characteristics of the ticket contained in the MessageEvent, the agent will look whether the ticket is already in the database *dbTickets*.

```
dbTickets.get(num,change.c2,change.t2,change.s2,change.b2,change.p2,
        change.l2,company))
```

If the ticket is present in the database, the operation is an update or a removal. Else, it is a addition of a new ticket. Anyway, the adequate operation is executed on the database *dbTickets*.

Finally, the whole treatment of the new data is triggered off, i.e. transforming into rules of type *'ticket->attribute'*, updating the DefaultTheory in order to eventually obtain other extensions, revising the content of the TheoryBase, ...

The History capability's plans

Event though we didn't have the time to implement completely this capability, we explain here what was its goal. The History capability is aimed to save time when the user issues a request that has already been made. In that case, the result of the request is just taken from the *dbHistoricResult* and there is no use to spawn a complete search. The plans used are *Save*P and *DeleteP*.

SaveP

This plan must be called each time a result of a search is obtained, i.e. a booking of a queuing of a ticket. The initial request (the general criteria's and the normal criteria's) are stored in the *dbHistoric* whereas the result of the search corresponding to these parameters are saved in the *dbHistoricResult*. A special field is used to differentiate the queued tickets and the booked ticket in the *dbHistoricResult*. Remember that to a search correspond zero, one or more queued tickets while there can't be more than one booked ticket.

Here is how the booked ticket is stored.

```
num3 = dbHistoricResult.nFacts();
dbBooked.get(1,c,tv,sm,b,p,h,comp);
dbHistoricResult.assert(num3+1,num1+1,c.as_string(),tv.as_string(), sm.as_string(),
        b.as_string(),p.as_int(),h.as_double(),comp.as_string(),"Booked");
```

And here is how the queued tickets are stored.

```
num3 = dbHistoricResult.nFacts();
for(int i=1;i<=dbQueued.nFacts();i++)
{
dbQueued.get(i,c2,t2,s2,b2,p2,h2,comp2);
dbHistoricResult.assert(num3+i,num1+1,c2.as_string(),t2.as_string(),s2.as_string(),
        b2.as_string(),p2.as_int(),h2.as_double(),comp2.as_string(),"Queued");
}
```

DeleteP

This plan is used to reinitialize the two historical databases, i.e. *dbHistoric* and *dbHistoricResult*. It is necessary when a ticket change occurs in the database of one CompanyAgent, at least if that change involves a ticket that has been requested by a TravelAgent. In that case, the saved results associated with the saved requests are distorted. Indeed, with the change of characteristic of one ticket, the result of the search could be different.

The other plans

Next to all these plans classified in capabilities, there are some plans of the TravelAgent that can't be attached to any capabilities. These are pure 'logistic' plans which don't have crucial functions. These plans are : *InitDatabasesP, NewCriteriasP, FillCriteresP, ViewP, InitSocialeDBP, AnalyseCriteriasP* and *EmptyBookQueueP*.

InitDatabasesP

This is the plan that initializes the database *dbDefaults*. Each default rule is repeated (with the number of the ticket in the rule) as many times as there are tickets in *dbTickets*. If we limit ourselves to the first default rule, the code is :

```
for (int i=1;i<=dbTickets.nFacts();i++)
        {
        dbDefaults.assert((6*(i-1))+1,"","-(ticket"+i+"->business)
        &-(ticket"+i+"->champagne)","ticket"+i+"->-champagne");
        }
```

NewCriteriasP

It handles the event *NewCriteriasE* that comes from the interface. That event contains the general criteria's (i.e. the departure and destination) specified by the user. All the plan has to do is filling the database *dbGeneralCriterias* with these criteria's.

```
dbGeneralCriterias.assert(dbGeneralCriterias.nFacts()+1,cri.dep, cri.des);
```

FillCriteresP

This plan has quite the same function as *NewCriteriasP* but this one fills the database *dbCriteres* with the 6 criteria's ordered by the user. The first field of *dbCriteres* is the name of

the criteria and the second one is the degree of preference that the user has associated with that criteria.

```
dbCriteres.assert(1,"champagne",fill.c);
dbCriteres.assert(2,"tv",fill.t);
dbCriteres.assert(3,"smoker",fill.s);
dbCriteres.assert(4,"business",fill.b);
dbCriteres.assert(5,"price",fill.p);
dbCriteres.assert(6,"length",fill.l);
```

ViewP

This is the plan that is spawned when the View button of the interface containing the results of the search is pressed. It takes the booked or queued tickets (according to the selection of the user) from the corresponding databases and displays the information on the screen.
For example, here is the code for the booked ticket (if there is one) :

```
dbBooked.get(1,ca,te,sm,bu,p,h,comp);
        interf.viewTickets(0,comp.as_string(),"date",ca.as_string(),te.as_string(),
        sm.as_string(),bu.as_string(),String.valueOf(p.as_int()).toString(),
        String.valueOf(h.as_double()).toString());
```

InitSocialeDBP

This very simple plan fills the database *dbSociale* with the names of the different airlines.

```
dbSociale.assert(1,"Virgin");
dbSociale.assert(2,"KLM");
dbSociale.assert(3,"Sabena");
```

AnalyseCriteriasP

This plan is called by the interface when all the criteria's have been specified. In order to go on with the treatment of these data, the agent has to know if the criteria's are the first specified or if the user has already spawned a search and wants now to modify the criteria's. Indeed, in the first case, the normal procedure must be followed (i.e. forming rules, filling the TheoryBase, ...). In the second case, the only thing to do is revising the belief concerning the criteria's. At least if the modifications involve preference criteria's. Indeed, if the general criteria's (departure, destination) have changed, the agent has to start from scratch and ask other tickets to the CompanyAgents.

EmptyBookQueueP

This plan is spawned when the agent wants to empty the databases *dbBooked*, *dbQueued* and *TicketBest*.

The CompanyAgent's plans

The CompanyAgent is less complex than the TravelAgent and hence has less plans. Here are the plans it uses : *InitDataCompanyP, FillTableP, AskTicketP, AskVacancyP, GetP, UpdateP,*

RemoveP. The two first plans are used for the initialization of the tickets database and of the corresponding interface. *AskTicketP* and *AskVacancyP* concern the communication with the *TravelAgents*. *AskTicketP* sends tickets to the TravelAgent while *AskVacancyP* tells that TravelAgent whether or not there is vacancy for a given ticket. Finally, *GetP*, *UpdateP* and *RemoveP* are used when a change occurs in the tickets database of the CompayAgent. These three plans are called from the interface because the administrator of the *dbMyTickets* has to introduce the changes via the interface.

InitDataCompanyP

This is a plan used by the CompanyAgents to fill their database of tickets (*dbMyTikets*) once these agents are created. In order to fill the databases with different tickets for each CompanyAgent even though an occurrence of the same plan is used by these CompanyAgents, a condition on the name of the agent that executes the plan is used.

```
String name = compano.getName();
```

Now, the database can be filled according to that name. For example, we put 3 tickets in the database of the Sabena airline.

```
if (name=="Sabena")
{
 dbMyTickets.assert(1,"Brussel", "Amsterdam","champagne","","smoker",
                "business",7000,7.2,0);
 dbMyTickets.assert(2,"London", "Paris","-champagne","-tv","","-business",
                8200,2.0,10);
 dbMyTickets.assert(3,"Paris", "Sydney","","","-smoker","-business",6000,5.0,2);
}
```

Once this is done, the plan posts the event that will trigger off the plan *FillTableP*.

```
@subtask(fill.post());
```

FillTableP

The only thing the plan does is filling the tables of the interface of the agent with the data contained in the database *dbMyTickets*.

AskTicketP

This plan handles the *AskTicketE* that has been sent by a TravelAgent (which is looking for tickets corresponding to the general criteria's specified by its user). It replies by sending to the TravelAgent that issued the request all the tickets he has in his database that corresponds to the general criteria's contained in the MessageEvent *AskTicketE*.

```
for (Cursor cur = dbMyTickets.get(i,asking.departu, asking.destinati,
      s1,s2,s3,s4,in,d,v);cur.next(); )
      {
      @send(company, recei.received(s1.as_string(),s2.as_string(),
            s3.as_string(),s4.as_string(),in.as_int(),d.as_double()));
```

}

And to conclude the transmission, it sends a 'closure' message that tells the TravelAgent that he has finished to send his tickets.

@send(company, all.end());

Let's note that the request of the TravelAgent is recorded in the database *dbCalls*, which is a database of the CompanyAgent. That database is useful when a characteristic of a ticket has changed. Indeed, the CompanyAgent has to know which TravelAgents have asked him the ticket that has been changed in order to warn them of the change.

dbCalls.assert(dbCalls.nFacts()+1,asking.departu,asking.destinati,company);

AskVacancyP

In this plan, the CompanyAgent answers to a TravelAgent that has asked him whether there was vacancy for a determined ticket. The first thing to do for the CompanyAgent is looking in his database the content of the field 'vacancy' of the ticket specified in the event *AskVacancyE*.

dbMyTickets.get(i,ask.dep,ask.dest,ask.ch,ask.tv,ask.sm,ask.bu,ask.pri,
ask.len,vacancy);

Than, it sends his reply to the TravelAgent that issued the MessageEvent *AskVacancyE*.

@send(company,reply.post(ask.ch,ask.tv,ask.sm,ask.bu,ask.pri,ask.len,vacant));

GetP

The plan *GetP* is triggered off from the interface. It searches the selected ticket (its number is specified in the event *GetE*) in the database *dbMyTickets* and displays its characteristics in the text fields of the interface.

RemoveP

First it sends to the TravelAgents interested by this ticket (i.e. that have requested it sooner) a MessageEvent *ChgeTicketE* containing the new characteristics (i.e. empty strings) and the old characteristics. See the section 2.6.1. (????) "How do we use belief revision ?" for deeper explanations.

@send(comp.as_string(),chgeTicket.post("",ch.as_string(),
"",tv.as_string(),"",sm.as_string(),
"",bu.as_string(),0,pr.as_int(),0.0,lh.as_double()));

Than, it removes the tickets from the database *dbMyTickets*.

And finally, it fills the tables of the interface by invoking the plan *FillTableP*.

@subtask(fill.post());

UpdateP

As in *RemoveP*, it sends a MessageEvent *ChgeTicketE* to the TravelAgents. In this case of an update, the new characteristics are not empty, but are those that has been specified in the text fields.

```
@send(comp.as_string(),chgeTicket.post(event.ch,ch.as_string(),
        event.tv,tv.as_string(),event.sm,sm.as_string(),
        event.bu,bu.as_string(),event.pr,pr.as_int(),event.lh,lh.as_double()));
```

It also records the changes in *dbMyTickets*.

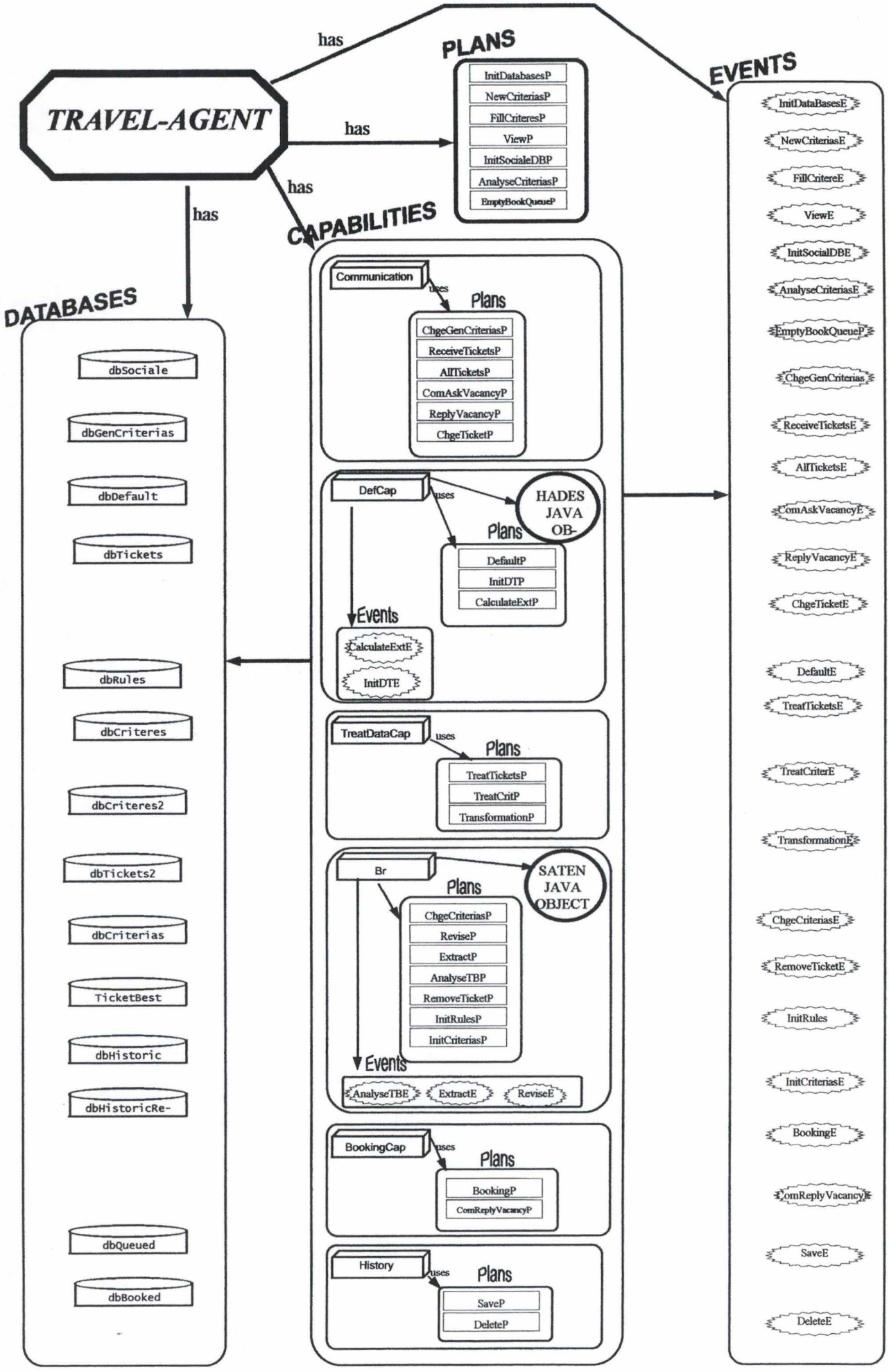
```
dbMyTickets.assert(event.in, event.depa,event.desti,event.ch,event.tv,event.sm,
        event.bu,event.pr,event.lh,event.va);
```

And finally, it spawns the filling of the table of the interface.

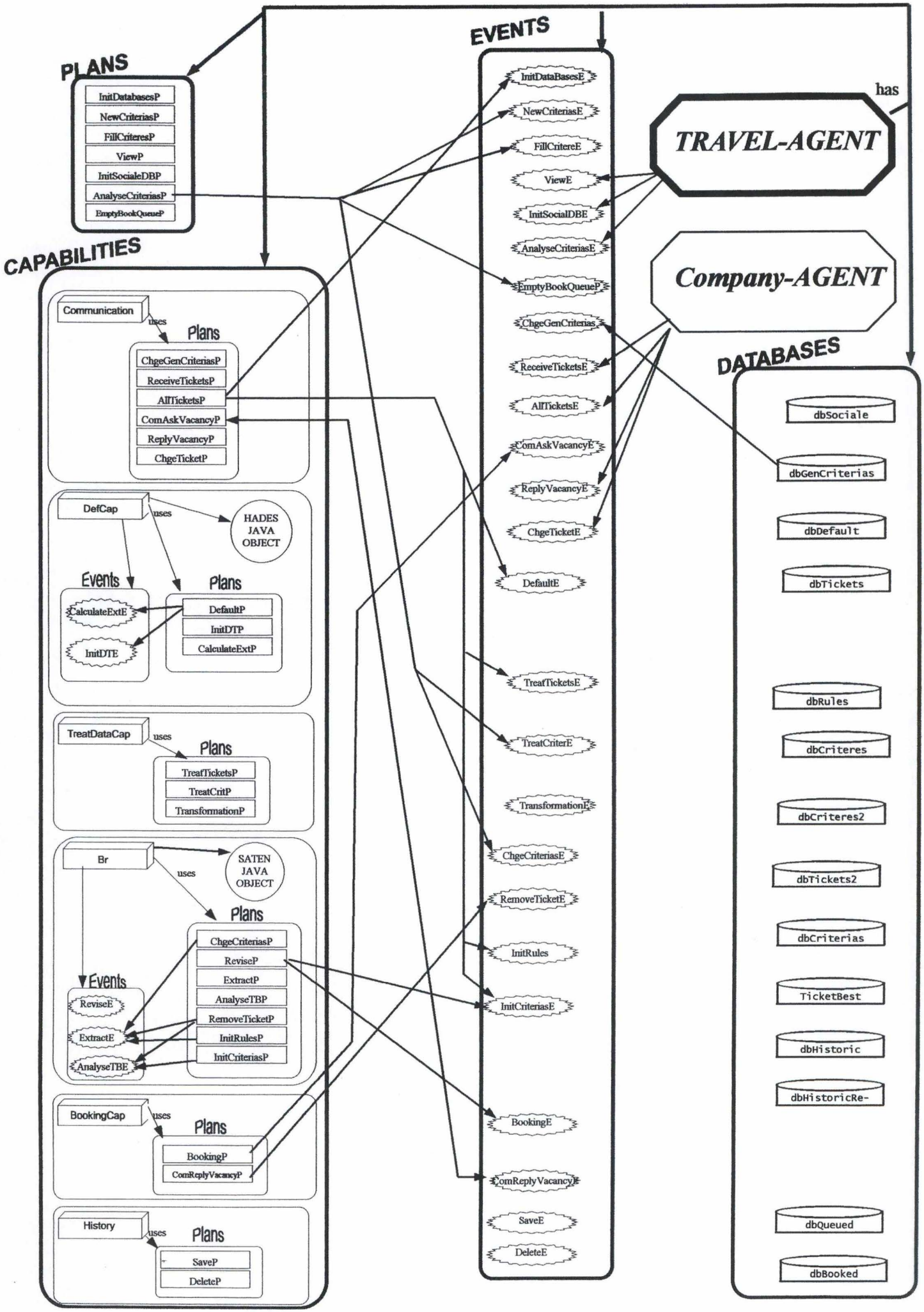
```
@subtask(fill.post());
```

***Annex 2* : Interconnection between capabilities, plans, events and databases**

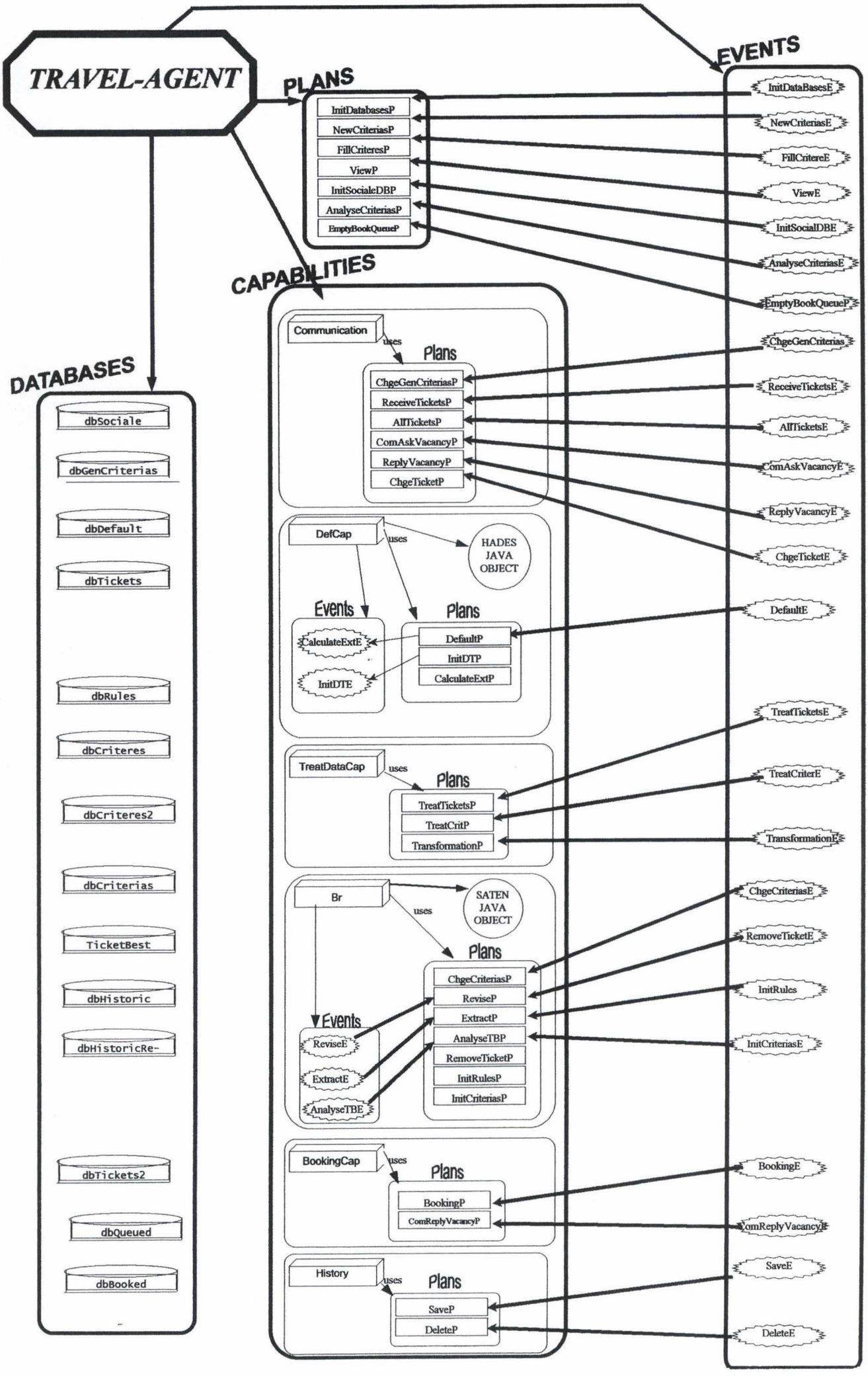
Interconnection between capabilities, plans, events and databases : General view


TRAVEL-AGENT
has
has
has
has
PLANS
InitDatabasesP
NewCriteriasP
FillCriteresP
ViewP
InitSocialeDBP
AnalyseCriteriasP
EmptyBookQueueP
EVENTS
InitDataBasesE
NewCriteriasE
FillCritereE
ViewE
InitSocialDBE
AnalyseCriteriasE
EmptyBookQueueP
ChgeGenCriterias
ReceiveTicketsE
AllTicketsE
ComAskVacancyE
ReplyVacancyE
ChgeTicketE
DefaultE
TreatTicketsE
TreatCriterE
TransformationE
ChgeCriteriasE
RemoveTicketE
InitRules
InitCriteriasE
BookingE
ComReplyVacancy
SaveE
DeleteE
CAPABILITIES
Communication
uses
Plans
ChgeGenCriteriasP
ReceiveTicketsP
AllTicketsP
ComAskVacancyP
ReplyVacancyP
ChgeTicketP
DefCap
uses
HADES JAVA OB-
Plans
DefaultP
InitDTP
CalculateExtP
Events
CalculateExtE
InitDTE
TreatDataCap
uses
Plans
TreatTicketsP
TreatCritP
TransformationP
Br
SATEN JAVA OBJECT
Plans
ChgeCriteriasP
ReviseP
ExtractP
AnalyseTBP
RemoveTicketP
InitRulesP
InitCriteriasP
Events
AnalyseTBE
ExtractE
ReviseE
BookingCap
uses
Plans
BookingP
ComReplyVacancyP
History
uses
Plans
SaveP
DeleteP
DATABASES
dbSociale
dbGenCriterias
dbDefault
dbTickets
dbRules
dbCriteres
dbCriteres2
dbTickets2
dbCriterias
TicketBest
dbHistoric
dbHistoricRe-
dbQueued
dbBooked

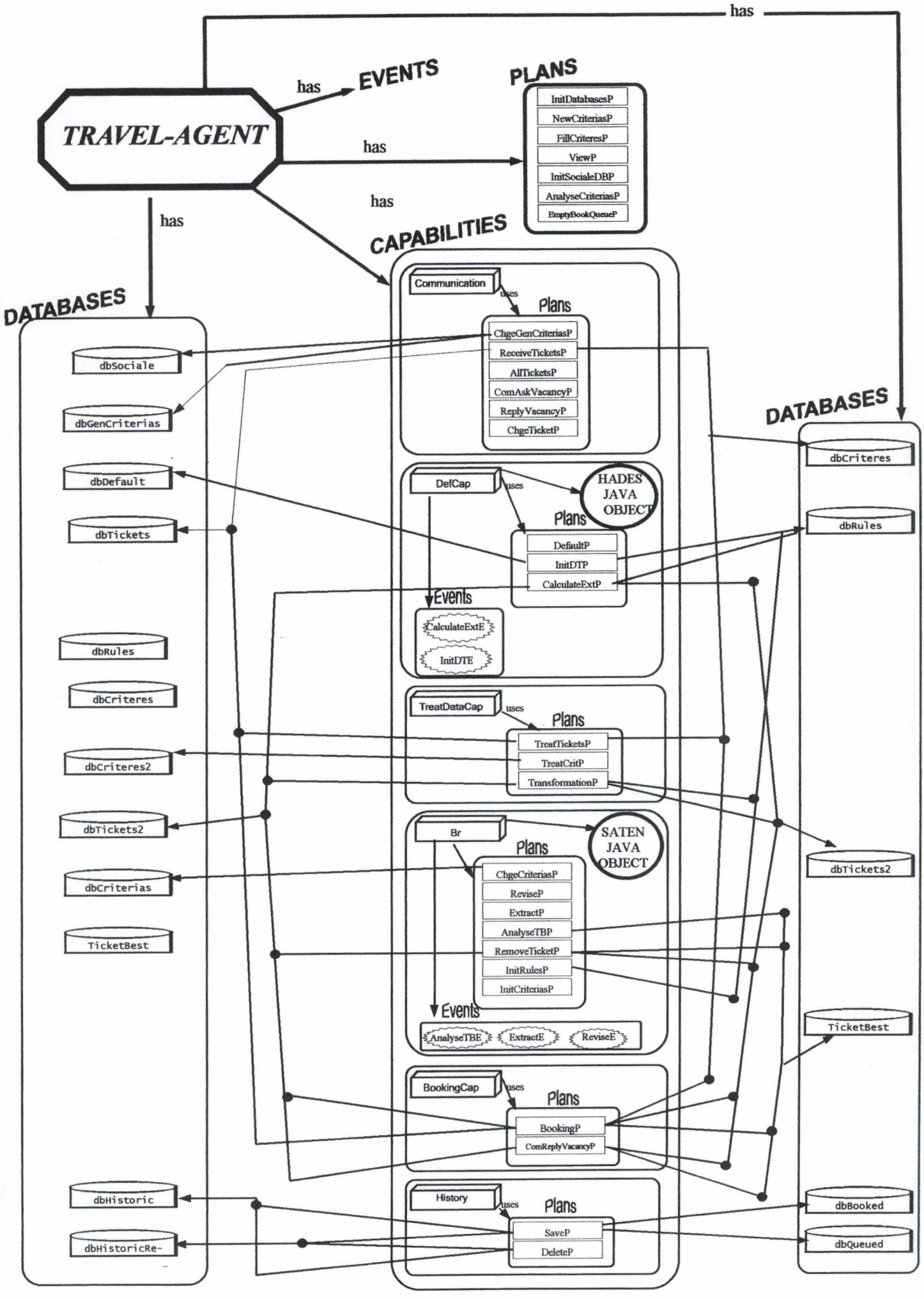
Interconnection between capabilities, plans, events and databases : The posting of the events


PLANS
InitDatabasesP
NewCriteriasP
FillCriteresP
ViewP
InitSocialeDBP
AnalyseCriteriasP
EmptyBookQueueP
EVENTS
InitDataBasesE
NewCriteriasE
FillCritereE
ViewE
InitSocialDBE
AnalyseCriteriasE
EmptyBookQueueP
ChgeGenCriterias
ReceiveTicketsE
AllTicketsE
ComAskVacancyE
ReplyVacancyE
ChgeTicketE
DefaultE
TreatTicketsE
TreatCriterE
TransformationE
ChgeCriteriasE
RemoveTicketE
InitRules
InitCriteriasE
BookingE
ComReplyVacancyE
SaveE
DeleteE
has
TRAVEL-AGENT
Company-AGENT
DATABASES
dbSociale
dbGenCriterias
dbDefault
dbTickets
dbRules
dbCriteres
dbCriteres2
dbTickets2
dbCriterias
TicketBest
dbHistoric
dbHistoricRe-
dbQueued
dbBooked
CAPABILITIES
Communication
uses
Plans
ChgeGenCriteriasP
ReceiveTicketsP
AllTicketsP
ComAskVacancyP
ReplyVacancyP
ChgeTicketP
DefCap
uses
HADES JAVA OBJECT
Events
Plans
CalculateExtE
InitDTE
DefaultP
InitDTP
CalculateExtP
TreatDataCap
uses
Plans
TreatTicketsP
TreatCritP
TransformationP
Br
uses
SATEN JAVA OBJECT
Plans
Events
ReviseE
ExtractE
AnalyseTBE
ChgeCriteriasP
ReviseP
ExtractP
AnalyseTBP
RemoveTicketP
InitRulesP
InitCriteriasP
BookingCap
uses
Plans
BookingP
ComReplyVacancyP
History
uses
Plans
SaveP
DeleteP

Interconnection between capabilities, plans, events and databases : The treatment of the events


TRAVEL-AGENT
PLANS
EVENTS
InitDatabasesP
NewCriteriasP
FillCriteresP
ViewP
InitSocialeDBP
AnalyseCriteriasP
EmptyBookQueueP
InitDataBasesE
NewCriteriasE
FillCritereE
ViewE
InitSocialDBE
AnalyseCriteriasE
EmptyBookQueueP
CAPABILITIES
DATABASES
Communication
uses
Plans
ChgeGenCriteriasP
ReceiveTicketsP
AllTicketsP
ComAskVacancyP
ReplyVacancyP
ChgeTicketP
ChgeGenCriterias
ReceiveTicketsE
AllTicketsE
ComAskVacancyE
ReplyVacancyE
ChgeTicketE
DefCap
uses
HADES JAVA OBJECT
Events
Plans
CalculateExtE
InitDTE
DefaultP
InitDTP
CalculateExtP
DefaultE
TreatDataCap
uses
Plans
TreatTicketsP
TreatCritP
TransformationP
TreatTicketsE
TreatCriterE
TransformationE
Br
uses
SATEN JAVA OBJECT
Plans
Events
ReviseE
ExtractE
AnalyseTBE
ChgeCriteriasP
ReviseP
ExtractP
AnalyseTBP
RemoveTicketP
InitRulesP
InitCriteriasP
ChgeCriteriasE
RemoveTicketE
InitRules
InitCriteriasE
BookingCap
uses
Plans
BookingP
ComReplyVacancyP
BookingE
ComReplyVacancyE
History
uses
Plans
SaveP
DeleteP
SaveE
DeleteE
dbSociale
dbGenCriterias
dbDefault
dbTickets
dbRules
dbCriteres
dbCriteres2
dbCriterias
TicketBest
dbHistoric
dbHistoricRe-
dbTickets2
dbQueued
dbBooked

Interconnection between capabilities, plans, events and databases : The access to the databases


has
TRAVEL-AGENT
has
EVENTS
has
PLANS
InitDatabasesP
NewCriteriasP
FillCriteresP
ViewP
InitSocialeDBP
AnalyseCriteriasP
EmptyBookQueueP
has
has
CAPABILITIES
DATABASES
Communication
uses
Plans
ChgeGenCriteriasP
ReceiveTicketsP
AllTicketsP
ComAskVacancyP
ReplyVacancyP
ChgeTicketP
dbSociale
dbGenCriterias
dbDefault
dbTickets
DefCap
uses
HADES
JAVA
OBJECT
Plans
DefaultP
InitDTP
CalculateExtP
Events
CalculateExtE
InitDTE
dbRules
dbCriteres
dbCriteres2
dbTickets2
dbCriterias
TicketBest
TreatDataCap
uses
Plans
TreatTicketsP
TreatCritP
TransformationP
Br
SATEN
JAVA
OBJECT
Plans
ChgeCriteriasP
ReviseP
ExtractP
AnalyseTBP
RemoveTicketP
InitRulesP
InitCriteriasP
Events
AnalyseTBE
ExtractE
ReviseE
BookingCap
uses
Plans
BookingP
ComReplyVacancyP
dbHistoric
dbHistoricRe-
History
uses
Plans
SaveP
DeleteP
DATABASES
dbCriteres
dbRules
dbTickets2
TicketBest
dbBooked
dbQueued

Annex 3 : A survey of agent construction tools

Commercial Products

AgentBuilder®	Brief Summary	AgentBuilder® Site	Reticular Systems, Inc.	Java	Integrated Agent and Agency Development Environment
AgenTalk	Brief Summary	AgenTalk Site	NTT	LISP	Multi-agent Coordination Protocols
Agentx	Brief Summary	Agentx Site	International Knowledge Systems	Java	Agent Development Environment
Aglets	Brief Summary	Aglets Site	IBM Japan	Java	Mobile Agents
Concordia	Brief Summary	Concordia Site	Mitsubishi Electric	Java	Mobile Agents
DirectIA SDK	Brief Summary	DirectIA SDK Site	MASA Group	C++	Adaptive Agents
Gossip	Brief Summary	Gossip Site	Tryllian	Java	Mobile Agents
Grasshopper	Brief Summary	Grasshopper Site	IKV++	Java	Mobile Agents
iGEN™	Brief Summary	iGEN™ Site	CHI Systems	C/C++	Cognitive Agent Toolkit
Intelligent Agent Factory	Brief Summary	Intelligent Agent Factory Site	Bits & Pixels	Java	Agent Development Tool
Intelligent Agent Library	Brief Summary	Intelligent Agent Library Site	Bits & Pixels	Java	Agent Library
JACK Intelligent Agents	Brief Summary	JACK Intelligent Agents Site	Agent Oriented Software Pty. Ltd.	JACK Agent Language	Agent Development Environment
JAM	Brief Summary	JAM Site	Intelligent Reasoning Systems	Java	Agent Architecture
Jumping Beans	Brief Summary	Jumping Beans Site	Ad Astra Engineering, Inc.	Java	Mobile Components
LiveAgent	Brief Summary	LiveAgent Site	Alcatel	Java	Internet Agent Construction
MadKit	Brief Summary	MadKit Site	Madkit Development Group	Java, Scheme, Jess	Multiagent Development Tool
Microsoft Agent	Brief Summary	Microsoft Agent Site	Microsoft Corporation	Active X	Interface creatures
Network Query Language	Brief Summary	Network Query Language Site	NQL Solutions		Programming Language
Pathwalker	Brief Summary	Pathwalker Site	Fujitsu	Java	Agent-oriented programming library
Swarm	Brief Summary	Swarm Site	Swarm Development Group	Objective C, Java	Multiagent Simulation
UMPRS	Brief Summary	UMPRS Site	Intelligent Reasoning Systems	C++	Agent Architecture
Via: Versatile Intelligent Agents	Brief Summary	Via Site	Kinetoscope	Java	Agent Building Blocks
Voyager	Brief Summary	Voyager Site	Object Space	Java	Agent-Enhanced ORB

Academic and Research Projects

Agent Building Shell (ABS)	Brief Summary	ABS Site	University of Toronto	COOrdination Language (COOL)	Agent Architecture
Agent Factory	Brief Summary	Agent Factory Site	University College Dublin, Ireland	Smalltalk-80	Agent Prototyping Environment
D'Agents	Brief Summary	D'Agents Site	Dartmouth University		Retrieval, Organization, and Presentation
Agent Tcl	Brief Summary	Agent TCL Site	Dartmouth University	Tcl	Mobile Agents
Architecture type-based Development Environment (ADE)	Brief Summary	ADE Site	University of Potsdam Dept. of Computer Science Professorship of Software Engineering TAXT	Java	Platform and Application Independent Agent Development Environment
Bee-gent	Brief Summary	Bee-gent Site	Toshiba Corporation Systems and Software Research Laboratories	Java	Software Development Framework
Bond Distributed Object System	Brief Summary	Bond Site	Purdue University	Java	Agent Framework
Cable	Brief Summary	Cable Site	Logica Corporation	Agent Definition Language, C++	System Architecture
DECAF Agent Framework	Brief Summary	DECAF Site	University of Delaware	Java	Agent Framework
dMARS	Brief Summary	dMARS Site	Australian Artificial Intelligence Institute Ltd.	C, C++	Agent-oriented Development and Implementation Environment
EXCALIBUR	Brief Summary	EXCALIBUR Site	GMD First, Technical University of Berlin		Autonomous Agent Architecture in a Complex Computer Game Environment
GenA	Brief Summary	GenA Site	CRIM	GenA	Mobile agent platform
Gypsy	Brief Summary	Gypsy Site	Technical University of Vienna	Java	Mobile Agents
Hive	Brief Summary	Hive Site	The Media Lab Massachusetts Institute of Technology	Java	Toolkit for Building Distributed Systems
InfoSleuth	Brief Summary	InforSleuth Site	MCC	Java	Collaborating agent environment
Infospiders	Brief	Infospiders	University of California	unknown	Adaptive Distributed

	Summary	Site	San Diego - Computer Science Dept.		Information Agents
JADE	Brief Summary	JADE Site	CSELT S.p.A., University of Parma	Java	Multiagent Framework
JAFMAS	Brief Summary	JAFMAS Site	University of Cincinnati	Java	Multiagent Framework
JATLite	Brief Summary	JATLite Site	Stanford University	Java	Java Packages for Multiagents
JATLiteBean	Brief Summary	JATLiteBean Site	University of Otago	Java	JavaBean Component
JIAC	Brief Summary	JIAC Site	Technische Universitat Berlin	Java	Agent Architecture
Kasbah	Brief Summary	Kasbah Site	Massachusetts Institute of Technology	unknown	Agent-mediated Electronic Commerce
KLAIM	Brief Summary	KLAIM Site	Universita' Di Firenze	Klaim	Kernel Language for Agent Interaction and Mobility
Knowbot® System Software	Brief Summary	Knowbot Site	CNRI	Python	Mobile Agents
Mobiware Middleware Toolkit	Brief Summary	Mobiware Site	Columbia University	Java	Mobile Networking Environment
MOLE	Brief Summary	MOLE Site	University of Stuttgart	Java	Mobile Agents
Multi-Agent Modeling Language (MAML)	Brief Summary	MAML Site	Central European University	MAML	Programming Language
MultiAgent Systems Tool (MAST)	Brief Summary	MAST Site	Technical University of Madrid	C++	Multiple Heterogeneous Agents
Open Agent Architecture™	Brief Summary	Open Agent Architecture Site	SRI International	C, C++, Prolog, Perl, Lisp, Java	Agent Framework
ProcessLink	Brief Summary	ProcessLink Site	Stanford University	unknown	Agent-based Framework
RETSINA	Brief Summary	RETSINA Site	Carnegie Mellon University		Communicating Agents
Social Interaction Framework (SIF)	Brief Summary	SIF Site	DFKI (German Research Institute for AI)	Java	Multi-agent System Toolkit
Sodabot	Brief Summary	Sodabot Site	MIT Artificial Intelligence Lab	unknown	Software Agent User-environment and Construction System
SOMA (Secure and Open Mobile Agent)	Brief Summary	SOMA Site	University of Bologna	Java	Agent Programming Environment

TeamBots	Brief Summary	TeamBots Site	The Robotics Institute Carnegie Mellon University	Java	Multiagent Mobile Robotics
TuCSoN	Brief Summary	TuCSoN Site	Universita di Bologna		Model For the Coordination of Internet Agents
Zeus	Brief Summary	Zeus Site	British Telecommunications Labs	Java	Agent Building Environment

Annexe 4 : An example of TheoryBase

An example of TheoryBase before extraction

ticket1|ticket2|ticket3	0.9999
ticket1->-smoker	0.9998
ticket1->-tv	0.9998
ticket1->business	0.9998
ticket1->-champagne	0.9998
ticket2->smoker	0.9998
ticket2->tv	0.9998
ticket2->business	0.9998
ticket2->-champagne	0.9998
ticket3->smoker	0.9998
ticket3->-tv	0.9998
ticket3->-champagne	0.9998
ticket3->business	0.9998
ticket1->-length 1	0.9998
ticket1->-length 2	0.9998
ticket1->-length 3	0.9998
ticket1->-length 4	0.9998
ticket1->length 5	0.9998
ticket2->length 1	0.9998
ticket2->-length 2	0.9998
ticket2->-length 3	0.9998
ticket2->-length 4	0.9998
ticket2->-length 5	0.9998
ticket3->-length 1	0.9998
ticket3->length 2	0.9998
ticket3->-length 3	0.9998
ticket3->-length 4	0.9998
ticket3->-length5	0.9998
ticket1->price1	0.9998
ticket1->-price2	0.9998
ticket1->-price3	0.9998
ticket1->-price4	0.9998
ticket1->-price5	0.9998
ticket2->-price1	0.9998
ticket2->price2	0.9998
ticket2->-price3	0.9998
ticket2->-price4	0.9998
ticket2->-price5	0.9998
ticket3->-price1	0.9998
ticket3->-price2	0.9998
ticket3->-price3	0.9998
ticket3->-price4	0.9998
ticket3->price5	0.9998
price1	0.9
length1	0.8

price2	0.72
length2	0.64
smoker	0.6
price3	0.54
length3	0.48
tv	0.4
price4	0.36
length4	0.32
business	0.3
champagne	0.2
price5	0.18
length5	0.16
ticket1	0.0001
ticket2	0.0001
ticket3	0.0001

The same TheoryBase after extraction

ticket1|ticket2|ticket3	0.9999
ticket1->-smoker	0.9998
ticket1->-tv	0.9998
ticket1->business	0.9998
ticket1->-champagne	0.9998
ticket2->smoker	0.9998
ticket2->tv	0.9998
ticket2->business	0.9998
ticket2->-champagne	0.9998
ticket3->smoker	0.9998
ticket3->-tv	0.9998
ticket3->-champagne	0.9998
ticket3->business	0.9998
ticket1->-length 1	0.9998
ticket1->-length 2	0.9998
ticket1->-length 3	0.9998
ticket1->-length 4	0.9998
ticket1->length 5	0.9998
ticket2->length 1	0.9998
ticket2->-length 2	0.9998
ticket2->-length 3	0.9998
ticket2->-length 4	0.9998
ticket2->-length 5	0.9998
ticket3->-length 1	0.9998
ticket3->length 2	0.9998
ticket3->-length 3	0.9998
ticket3->-length 4	0.9998
ticket3->-length5	0.9998
ticket1->price1	0.9998
ticket1->-price2	0.9998
ticket1->-price3	0.9998
ticket1->-price4	0.9998
ticket1->-price5	0.9998
ticket2->-price1	0.9998
ticket2->price2	0.9998
ticket2->-price3	0.9998
ticket2->-price4	0.9998
ticket2->-price5	0.9998
ticket3->-price1	0.9998
ticket3->-price2	0.9998
ticket3->-price3	0.9998
ticket3->-price4	0.9998
ticket3->price5	0.9998
price1	0.9

business	0.3
length5	0.16
ticket1	0.0001

Annexe 5 : An example of DefaultTheory

An example of default theory

ticket1->-champagne
ticket1->business
ticket1->price1
ticket1->-price2
ticket1->-price3
ticket1->-price4
ticket1->-price5
ticket1->-length1
ticket1->-length2
ticket1->-length3
ticket1->-length4
ticket1->length5
ticket2->champagne
ticket2->-tv
ticket2->smoker
ticket2->-business
ticket2->-price1
ticket2->-price2
ticket2->-price3
ticket2->-price4
ticket2->price5
ticket2->length1
ticket2->-length2
ticket2->-length3
ticket2->-length4
ticket2->-length5

-(ticket1->business)&-(ticket1->champagne)ticket1->-champagne
-(ticket1->business)&-(ticket1->tv)ticket1->-tv
-(ticket1->business)&-(ticket1->smoker)ticket1->-smoker
(ticket1->business)-(ticket1->-champagne)ticket1->champagne
(ticket1->business)-(ticket1->-tv)ticket1->tv
(ticket1->business)-(ticket1->-smoker)ticket1->smoker
-(ticket2->business)&-(ticket2->champagne)ticket2->-champagne
-(ticket2->business)&-(ticket2->tv)ticket2->-tv
-(ticket2->business)&-(ticket2->smoker)ticket2->-smoker
(ticket2->business)-(ticket2->-champagne)ticket2->champagne
(ticket2->business)-(ticket2->-tv)ticket2->tv
(ticket2->business)-(ticket2->-smoker)ticket2->smoker